© 2021 Ali Kheradmand

FOUNDATIONS FOR PRACTICAL NETWORK VERIFICATION

BY

ALI KHERADMAND

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Brighten Godfrey, Chair
Professor Grigore Rosu
Assistant Professor Tianyin Xu
Dr. Jitendra Padhye, Microsoft Research

Abstract

Computer networks are large and complex and the often manual process of configuring such systems is error-prone, leading to network outages and breaches.This has ignited research into network verification tools that given a set of operator intents, automatically check whether the configured network satisfies the intents. In this dissertation, we argue that existing works in this area have important limitations that prevent their widespread adoption in the real world. We set to address these limitations by revisiting the main aspects of network verification: verification framework, intent specification, and network modeling.

First, we develop #PEC, a symbolic packet header analysis framework that resolves the tension between expressiveness and efficiency in previous works. We provide an extensible library of efficient match-types that allows encoding and analyzing more types of forwarding rules (e.g. Linux iptables) compared to most previous works. Similar to the state-of-the-art, #PEC partitions the space of packet headers into a set of equivalence classes (PECs) before the analysis. However, it uses a lattice-based approach to do so, refraining from using computationally expensive negation and subtraction operations. Our experiments with a broad range of real-world datasets show that #PEC is $10\times$ faster than similarly expressive state-of-the-art. We also demonstrate how empty PECs in previous works lead to unsound/incomplete analysis and develop a counting-based method to eliminate empty PECs from #PEC that outperforms baseline approaches by $10 - 100\times$.

Next, we note that network verification requires formal specifications of the intents of the network operator as a starting point, which are almost never available or even known in a complete form. We mitigate this problem by providing a framework to utilize existing low-level network behavior to infer the high-level intents. We design Anime, a system that given observed packet forwarding behavior, mines a compact set of possible intents that best describe the observations. Anime accomplishes this by applying optimized clustering algorithms to a set of observed network paths, encoded using path features with hierarchical values that yield a way to control the precision-recall tradeoff. The resulting inferred intents can be used as input to verification/synthesis tools for continued maintenance. They can also be viewed as a summary of network behavior, and as a way to find anomalous behavior. Our experiments, including data from an operational network, demonstrate that Anime produces higher quality (F-score) intents than past work, can generate compact summaries with minimal loss of precision, is resilient to imperfect input and policy changes, scales to large networks, and finds actionable anomalies in an operational network.

Finally, we turn our attention to modeling networking devices. We envision basing data plane analysis on P4 as the modeling language. Unlike most tools, we believe P4 analysis must be based on a precise model of the language rather than its informal specification. To this end, we develop a formal operational semantics of the P4 language during the process of which we have identified numerous issues with the design of the language. We then provide a suite of formal analysis tools derived directly from our semantics including an interpreter, a symbolic model checker, a deductive program verifier, and a program equivalence checker. Through a set of case studies, we demonstrate the use of our semantics beyond just a reference model for the language. This includes applications for the detection of unportable code, state-space exploration, search for bugs, full functional verification, and compiler translation validation.

Thesis Statement: Existing network (data plane) verification tools have fundamental shortcomings in their efficiency, expressiveness, fidelity, or ease of use that limit their real-world practicality. It is possible to address these shortcomings by developing new foundations for the main ingredients of network verification, namely system modeling, property specification, and verification data structures and algorithms.

Acknowledgments

As I approach the end of my long Ph.D. journey, I would like to thank a small subgroup of many people that have helped me reach this point.

First and foremost, I want to express my sincere gratitude to my advisor, Professor Brighten Godfrey, for his great support and guidance in the past four years of my journey. I have been fortunate to work with and learn from Brighten both in academia at the University of Illinois at Urbana-Champaign (UIUC) and in industry at Veriflow. Brighten is among a few people I know that are both extremely smart and extremely humble. I have learned a lot from him not only about high-quality scientific research and its transformation into industrial products, but also about proper execution, communication, presentation, and leadership. Brighten's uninterrupted support and positive attitude have always helped me carry through the ups and downs of my journey.

Next, I want to thank my co-advisor Professor Grigore Rosu for his support during the first years of my graduate studies, while I was adjusting to my Ph.D. life. Grigore's ability to motivate others has been a source of inspiration and his vision of semantic-based formal analysis is the basis for many works, including Chapter 4 in this dissertation.

I would also like to thank the other brilliant members of my Ph.D. committee: Professor Tianyin Xu and Dr. Jitendra Padhye, for accepting to be on the committee and for providing great feedback for this work.

Working at Fujitsu Laboratories of America introduced me to the world of network verification. I thank its director of research, Dr. Mukul Prasad, and my great colleague and friend, Dr. Alex Horn. I've learned a great deal about discipline from Alex and hours of mind-stimulating discussions with him have led to many ideas. So far, we have published only a small part of what we explored together including works inside and outside this dissertation. Particularly Chapter 2 is joint work with Alex. Among other contributions, I particularly thank him for his role in initiating the project. Also for identifying the problem of empty PECs in related work and his neat idea of using model-counting to detect them.

A huge shout-out to Veriflow (acquired by VMware), especially the members of its core team. It is rare to find opportunities to work on problems that are both interesting and useful and I thank Veriflow's leadership, especially Brighten, for providing me with such an opportunity. I enjoyed every second of my time at the company, and the work I've done there (a secret for now) is among the ones that I am most proud of. It helped me learn the difference between scientific research that only looks good on paper and the ones that

actually make a difference in the real world. I was lucky to be surrounded by such a smart and fun group of colleagues including Brighten and Drs. Santhosh Prabhu, Wenxuan Zhou, Mehedi Bakht, and Ahmed Khurshid. Particularly, I have worked closely with Santhosh both inside and outside Veriflow. This includes works on scalable data plane and control plane verification, which are not part of this dissertation. Santhosh is smart and a fast learner who knows how to get things done, which has been very inspiring to me. I have also benefited from his feedback on virtually all works in this dissertation, and I thank him for that.

Parts of the material in this dissertation are based upon works supported by the National Science Foundation under grant numbers CCF-1421575 and CNS-1513906. Part of the work in Chapter 2 was done while at Fujitsu Laboratories of America. I am also grateful for the Ray Ozzie Fellowship for partly supporting the first year of my Ph.D. studies.

Although not included in this dissertation, I'd like to acknowledge and thank my other great collaborators Professor Matthew Caesar, Kuan-Yen Chou, and Bingzhe Liu for works on control plane verification and extending verification beyond the networking layer.

I would also like to thank the K development team including the members of Runtime Verification Inc. and Formal Systems Laboratory for their help with the K framework. I also appreciate the support of Nate Foster and other members of the P4 Language Consortium.

UIUC and its Computer Science Department have been a wonderful second home to me. I appreciate the thriving atmosphere created by its faculty, staff, and students. Among many great staff, I'd like to thank Kara MacGregor, Maggie Chappell, and Viveka Kudaligama. Also many thanks to the awesome members of the networking group for numerous fun and interesting meetings and gatherings and for their feedback on works in this dissertation.

Life in Urbana-Champaign could not be as much fun without many good friends that made it feel like home. It would be impossible to name everybody here, but I want to particularly thank my small and closed social bubble that helped me endure the ongoing pandemic: Mohammad Amin, Rasoul, Happy, Sahand, Amin, and Hadi.

Above all, I want to thank my family. I am forever indebted to my amazing parents, Bozorg and Pari, for their wisdom and sacrifices and for raising me in a peaceful environment full of love that helped me grow into the person that I am today; my lovely sister and brother, Parastoo and Mohsen, who have always been supportive of me; and my dearest grandparents, who are the source of my energy. Finally, to my best friend and wonderful partner in crime, Farnaz, who has changed me in many positive ways and has always encouraged me to get up when I fell, I express my utmost love and gratitude. Looking forward to many more years of joy together. I couldn't have asked for a better family. These people are the most precious belongings in my life and to whom I dedicate this dissertation.

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Verification Framework: Expressive and Efficient Formal Network Analysis [1] 6
- 2.1 Background and Motivation 6
- 2.2 #PEC Framework 13
- 2.3 Evaluation 22
- 2.4 Conclusion 30

Chapter 3 Specification: Mining High-Level Intents from Low-Level Behavior [2] . . 32
- 3.1 Motivation 33
- 3.2 Anime Framework 36
- 3.3 Evaluation 45
- 3.4 Discussion 54
- 3.5 Conclusion 55

Chapter 4 Modeling: Formal Semantics of P4 and its Applications [3] 56
- 4.1 Background and Challenges 58
- 4.2 P4K 63
- 4.3 Evaluation 71
- 4.4 Applications 72
- 4.5 Conclusion 80

Chapter 5 Related Work 82
- 5.1 Network Verification 82
- 5.2 Facilitating Specification 82
- 5.3 Modeling and P4 Analysis 83

Chapter 6 Conclusion 86
- 6.1 Future Work 86

References 90

Chapter 1: Introduction

Modern-day computer networks are increasingly large and complex. Correct configuration and maintenance of such a complex system is an important concern for the organization operating it. In a typical organization today, given an informal (i.e. communicated in human language) high-level description of how the network should operate, a network administrator configures the network devices to achieve the high-level goals. These descriptions implicitly define what we refer to as *intents*. Intents are usually network-wide properties of the network forwarding behavior, e.g., "the traffic received from the Internet destined to IP x must reach node A"; "web traffic from node B to node C must go through a DPI device and be resilient to any single link failure"; or "devices in the region X should not be able to communicate with devices in the region Y".

The administrator converts these descriptions into low-level device configurations, often manually. Consequently, there is a significant gap between the high-level intents and the low-level configurations and subsequently the network behavior [4]. This gap is a source of many misconfigurations and network problems, leading to catastrophic consequences including network outages and breaches that often make news headlines [4].

These problems are the impetus behind, among other things, a significant academic and industrial push towards applying formal verification to computer networks with the goal of increasing network reliability (e.g., [5, 6, 7, 8, 9, 10, 11, 12]), a topic typically known in the community as *network verification* and can roughly be divided into two major categories: "Data plane verification" tools (e.g., [5, 7, 8, 13, 14]) analyze static snapshots of the data plane, while "control plane verification" (e.g., [10, 11, 12]) proactively ensures a network is free of latent bugs by analyzing the control logic under various environmental events[1]. In this dissertation, we will be focusing on data plane verification, although our contributions are useful for control plane verification as well.

As illustrated in Figure 1.1, formal verification of any system requires three main ingredients: a model of the system under consideration, formal specifications of properties of interest, and a verification method (algorithms and data structures) that can check if the model satisfies the properties. Existing network verification tools have notable limitations in these ingredients when it comes to the applicability of these tools in practice. In terms of verification algorithms and data structures, existing works focus either on efficiency (e.g., [5, 6, 7, 14]) or expressiveness (e.g., [8, 15]), at the cost of the other. Efficiency (ability to analyze large networks with modest time and memory overhead) and expressiveness (ability to support

[1] The boundary between the two categories is quite blurry especially in modern software-based networks.

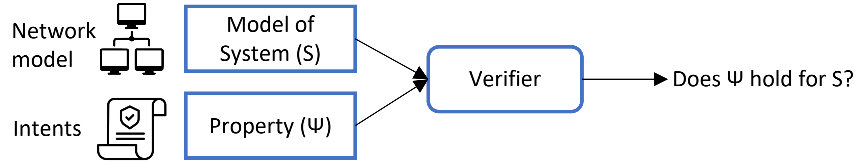


Figure 1.1: Formal verification process

various types of networking technologies) are both necessary for scaling verification to today's large and complex networks. In terms of ease of use, none of the existing work facilitates the process of intent specification for network operators. This is a particularly important limitation given that in the real-world, formal specifications of the intended behavior are rarely available or even known to the network operators. In terms of modeling, the majority of works develop their fixed models of network elements in an ad-hoc fashion that not only cannot be used with modern "programmable" network devices (e.g., [16]) but also does not yield practical means to examine and trust their fidelity. Even the works that do consider programmable networks (e.g., [17, 18, 19, 20, 21]) lack rigorous formal foundations or have it only partially (ignoring the more complex aspects of such networks). These limitations negatively affect the adoption of network verification technology in the real world.

The goal of this dissertation is to provide foundations for the three ingredients of formal network verification, addressing the aforementioned shortcomings of existing work that limit their practicality. Concretely, we make the following contributions:

- **Verification algorithms and data structures (Chapter 2)**

  At its core, network (data plane) verification can be viewed as symbolic execution over a specific domain in which network devices are viewed as functions that read and transform variables of a specific type, namely packet headers. Thus, there is a need for efficient and flexible data structures (and their associated algorithms) for symbolic representation and manipulation of packet headers. Existing works have evolved from using off-the-shelf SAT/SMT solvers (e.g., [15]) with poor scalability, to using domain-specific geometric models (e.g., [7]), and recently into more efficient state-of-the-art approaches (e.g., [5, 6, 8, 14]) that partition the space of packet headers into packet equivalence classes (PEC) with identical forwarding behavior *before* the start of the symbolic execution. While the idea of using PECs has generally increased the efficiency of network analysis [8], the choice of data structures to encode the forwarding rules and algorithm to perform the partitioning has notable implications on the efficiency and expressiveness of the resulting tool.

  The PEC-based approaches in existence today focus either on efficiency or expressiveness,

at the cost of the other. Specifically, APV [8] uses Binary Decision Diagrams (BDDs) to encode forwarding rules and construct the PECs. While flexible, BDDs been shown to incur significant overhead per packet header bit, performing poorly when analyzing large-scale data centers. This has prompted recent works to use more efficient representations to encode and manipulate packet sets, such as ranges (Delta-net [5], VeriFlow [6]) and Ternary Bit Vectors (ddNF [14]). However, these representations, while more compact and efficient, are strictly less expressive than BDDs, preventing them from encoding and analyzing various types of forwarding rules such as the ones in Linux iptables.

Our work, #PEC, resolves this tension between expressiveness and efficiency through a new PEC-construction approach. Instead of bounding ourselves to a specific representation, we provide an extensible library of efficient *match types* that one can mix and match to encode various parts of forwarding rules, allowing it to analyze more types of devices. Moreover, #PEC employs an efficient algorithm to organize rule match conditions in a Set Intersection Closed Lattice (SICL) and use it to define PECs in a way that spares it from using negation/subtraction operations, which are computationally more expensive. We also show that the existence of empty PECs can lead to wrong analysis results, and experimentally demonstrate such problems with ddNF on real-world datasets. Subsequently, we provide an efficient way to eliminate empty PECs (a coNP-hard problem) using a counting-based method that is at least an order of magnitude faster than naive SAT/BDD-based solutions.

Our experiments with a broad range of real-world datasets show that #PEC is 10× faster than APV. By achieving precision, expressiveness, and performance, our work answers a longstanding quest that has spanned multiple generations of formal network analysis techniques.

- **Property specification (Chapter 3)**

  Network verification tools require formal (i.e. defined in a precise language) specifications of the intended behavior as a starting point, which are almost never available or even known in a complete form. We discuss a novel approach to mitigate this problem by providing a framework to utilize existing low-level network behavior to infer the high-level intents. We design Anime, a system that given observed packet forwarding behavior, mines a compact set of possible intents that best describe the observations. The resulting inferred intents can be used as input to verification/synthesis tools for continued maintenance. They can also be viewed as a summary of network behavior, and as a way to find anomalous behavior.

We provide a setup to express both low-level behavior and high-level intents via their associated features. Each feature corresponds to one aspect of the behavior, including packet header information, devices along the path, and environmental conditions such as time of path observation, device or link state, etc. We design our features to have hierarchical values (like in CIDR), providing fine-grained control over the trade-off between precision and recall. We then use this setup to formally define intent inference as an NP-hard constrained cost optimization problem related to our quality and compactness measures. We heuristically solve the problem by grouping relevant behavior using clustering techniques with the cost function as a measure of dissimilarity and finding the most specific intent that represents all behavior of each group. We also develop a suite of highly effective optimizations, including indexing and parallelization, that allows our approach to scale to large networks with millions of forwarding paths.

Our experiments, including data from an operational network, demonstrate that Anime produces higher quality (F-score) intents than past work, can generate compact summaries with minimal loss of precision, is resilient to imperfect input and policy changes, scales to large networks, and finds actionable anomalies in an operational network.

- **System modeling (Chapter 4)**

  P4 [22] is a popular language for programmable packet processors with a relatively high expressive power. One could also view P4 as a standard modeling language for data plane behavior, not only for the emerging programmable hardware but also for the traditional fixed-function devices. We argue that formal analysis tools for any language, including P4, must be based on a formal semantics of the language rather than its informal specifications. To this end, we provide a formal operational semantics of the P4 language in K [23], a programming language semantics engineering framework based on term rewriting [24].

  Our formalization gives precise meaning to P4 programs and networks in terms of their operational behavior, which can serve as a reference model for verification tools. We faithfully formalized *all* language features mentioned in the official P4 language specification, with a few exceptions corresponding to features whose meaning was ambiguous or incorrect, or under-specified and we did not find any satisfactory way to correct it. We have reported some of these issues to the P4 language designers and made suggestions for fixing the issues.

  One of the important features of our semantics is its executability, i.e. we can test our work by executing P4 programs directly based on the semantics. This feature

enabled us to validate our semantics by executing test cases from an official P4 compiler front-end and an additional manually crafted test suite.

We also provide a suite of formal analysis tools derived directly from our formal semantics including an interpreter, a symbolic model checker, a deductive program verifier, and a program equivalence checker. Through a set of case studies, we demonstrate how our semantics and the derived tools can be used beyond just a reference model for the language. This includes applications for the detection of unportable code, state-space exploration, search for bugs, full functional verification, and compiler translation validation.

Chapters 2 to 4 are dedicated to the technical contributions. Chapter 5 overviews the related work. We conclude and discuss future research directions in Chapter 6.

Chapter 2: Verification Framework: Expressive and Efficient Formal Network Analysis [1]

This chapter overviews our verification framework. We first provide background on data plane verification and PEC-based network analysis. We identify the challenges and previous works' shortcomings in addressing them (§ 2.1). We then review our technical approach (#PEC) and the main insights in addressing the challenges (§ 2.2). Finally, § 2.3 details our experiments and case studies with real-world datasets and compares the results with the related work.

2.1 BACKGROUND AND MOTIVATION

Network data plane is comprised of networking devices (e.g., switches, routers, firewalls) that process and carry out the network traffic in form of data packets. We abstractly view the data plane as a set of *match-action tables*, connected to each other according to the network topology. For simplicity and without loss of generality we assume each device contains a single table, although the internal details of a packet processor are typically more complicated (Chapter 4). Each table contains multiple *forwarding rules* each comprised of two parts: a *match* part describing the conditions on the packet header for the rule to match on an incoming packet, and an *action* part that describes the action that would be performed on the packet in case the rule is matched. The possible actions include selecting output port, dropping the packet, or rewriting parts of the header[1]. The rules in a table are ordered according to their priority. When a packet arrives at a table, the highest priority rule that matches the packet header would be selected and the action part of the rule is applied to the packet. For instance, in the table shown in Figure 2.1b, the first rule matches on any packet with the IP destination in 10.0.0.0/17 whose IP protocol number is not UDP (symbol ! indicates negation). Say, a packet with the IP 10.0.0.0 and protocol UDP arrives at the device. The highest priority rule matching the packet header is rule 2. Therefore, the packet will be forwarded to d_2, according to the action of that rule.

The data plane is programmed (i.e., the tables are populated) by the control plane which in turn is configured or programmed according to the user's intents. Due to the scale and complexity of computer networks, it is extremely challenging to know if the configured network conforms to the user intents. This challenge is the source of many misconfigurations resulting in network outages and security breaches. These problems have ignited research

[1]For simplicity of presentation, we assume tables do not modify the packet headers in this chapter, though our framework does support packet modification using the same approach as described in [14].

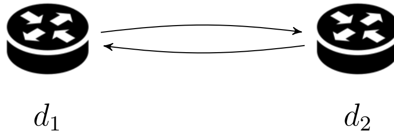
into automated verification techniques that given a network and the property of interest (specifications of user intents), can check whether the network conforms to the property. In this work, we focus on network data plane verification, which deals with analyzing a single snapshot of the data plane. However, the techniques introduced here are also applicable to network control plane verification, which deals with analyzing the control plane logic and configurations that program the data plane.

At its core, data plane verification can be viewed as a form of *symbolic execution* [25]. Symbolic execution is a well-known technique for finding problematic inputs for a program. A symbolic execution engine interprets a given program with partially or fully symbolic inputs. At each decision point, instead of taking a single branch, the engine takes all branches and considers the conditions of each branch on the symbolic inputs. Along each path, it accumulates all conditions and if it reaches an undesirable state, the engine would attempt to resolve the constraints: find values for the symbolic variables that satisfy the constraints. Software symbolic execution tools often use SAT/SMT [26] solvers for this purpose.

Data plane verification is essentially symbolic execution in a special domain. In this domain, the program is the set of match-action tables (priority ordered list of forwarding rules) and the topology that connects the tables. The input to this program is a packet header.

Consider the small network of Figure 2.1 consisting of two devices, d_1 and d_2. Say, we are interested in knowing if there are any packets entering d_1 that would experience a forwarding loop. To do so, we start with a symbolic packet entering d_1. For this packet to be forwarded to d_2 its header should not match the conditions of the first rule (otherwise it would be consumed by d_1), and it should match the conditions of the second rule. For d_2 to forward the packet back to d_1, the packet should match the condition of the first rule in d_2. Therefore, to know which packets (if any) would cause a forwarding loop, we need to resolve the following logical constraint $\Phi = \neg d_1.rule_1.match \wedge d_1.rule_2.match \wedge d_2.rule_1.match$.

The question is how one can encode and resolve these symbolic constraints. Following the standard practice in general software verification, early works in network verification (e.g., [15]) encoded the constraints as propositional or first-order logic formulas and used off-the-shelf SAT or SMT solvers to check their satisfiability. However, SAT/SMT-based approaches are not a proper fit for the domain of network verification. These solvers are optimized for finding a single satisfiable assignment. It will be easier for an operator to pinpoint the root of a problem in device configurations if all packets violating the desired property are provided to her. SAT/SMT solvers are not optimized for such purposes. More importantly, even for a single satisfiable assignment, these tools are not efficient enough to be used for analyzing large-scale networks with hundreds of thousands of devices and millions of forwarding rules.



(a) Topology of the tables.

Rule	Match		Action
	Destination	*Protocol*	
1	10.0.0.0/17	!UDP	Receive
2	10.0.0.0/16	{TCP,UDP}	Forward (d2)
3	Any	Any	Drop

(b) Match-action table of d_1.

Rule	Match		Action
	Destination	*Protocol*	
1	10.0.128.0/17	Any	Forward (d1)
2	Any	Any	Drop

(c) Match-action table of d_2.

Figure 2.1: A simple data plane used as the running example. The rules in the tables are sorted from highest to lowest priority.

This has sparked research into designing domain-customized approaches for encoding and handling constraints for symbolic packet headers, addressing the limitations of SAT/SMT solvers especially with regards to efficiency. The state-of-the-art uses the idea of Packet Equivalence Classes (PEC). By statically analyzing the rule tables, the PEC-based analysis tools (e.g., [5, 6, 8, 14]) partition the space of packet headers into a set of equivalence classes such that: (i) no two PECs overlap with each other, and (ii) all packets belonging to the same PEC would experience the same forwarding behavior throughout the network. Subsequently, such tools replace the complex forwarding rules on devices with significantly simpler ones that only match on PECs which essentially are treated as labels.

In the example above, one way of partitioning is visualized in Figure 2.2. In the figure, the horizontal and vertical axes correspond to the destination IP address and the IP protocol number of the packets. For simplicity of visualization, we omit the other dimensions of the packet header space corresponding to other header fields. Each numbered region in the figure corresponds to one PEC. For example, PEC_3 consists of UDP packets with destination IP in the range $[10.0.0.0, 10.0.128.0)$. Note that any two packets belonging to the same PEC, say a UDP packet with destination 10.0.0.1 and another UDP packet with destination 10.0.0.2 (both belonging to PEC_3), experience the same behavior throughout the network: both get forwarded from d_1 to d_2, where they get dropped.

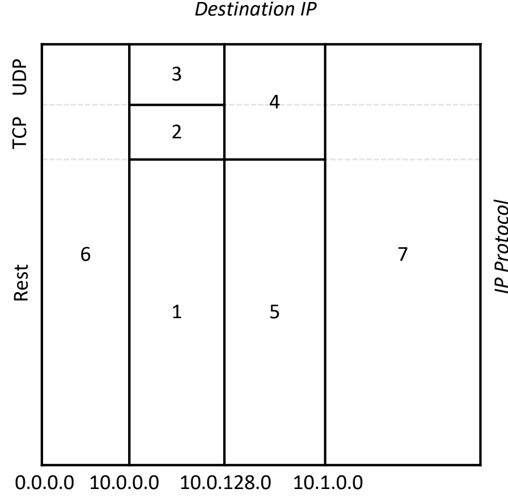


Figure 2.2: One way of defining PECs for the example in Figure 2.1.

Match	*Action*
PEC_1	Receive
PEC_2	Receive
PEC_3	Forward (d2)
PEC_4	Forward (d2)
PEC_5	Drop
PEC_6	Drop
PEC_7	Drop

(a) PEC-based equivalent of d_1.

Match	*Action*
PEC_1	Drop
PEC_2	Drop
PEC_3	Drop
PEC_4	Forward (d1)
PEC_5	Forward (d1)
PEC_6	Drop
PEC_7	Drop

(b) PEC-based equivalent of d_2.

Figure 2.3: PEC-based equivalent of tables in our running example according to the PEC definition in Figure 2.2.

Given this particular partitioning, we can replace the tables in Figure 2.1 with the their PEC-based equivalents in Figure 2.3. From this point, the analysis can be performed based on these labels. For instance, to find out if there are any forwarding loops starting from d_1, we can simply intersect the set of PECs that are forwarded from d_1 to d_2 ($\{PEC_3, PEC_4\}$) with the set of PECs that are forwarded from d_2 to d_1. ($\{PEC_4, PEC_5\}$). Performing this set intersection is significantly more efficient than encoding Φ (introduced above) as a propositional or first-order logic formula and using SAT/SMT solvers to check its satisfiability. Also note that in contrast to the SAT/SMT-based approach that produces a single packet, the intersection result ($\{PEC_4\}$) encompasses the set of all packets that experience the forwarding loop.

We emphasize that Figure 2.2 illustrates just one among many possible ways to define PECs for our example. For instance, PEC_6 and PEC_7 could be combined into a single PEC.

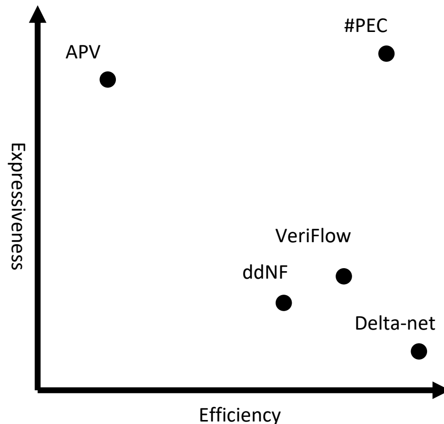


Figure 2.4: Landscape of PEC-based analysis tools wrt. efficiency and expressiveness.

Source	*Destination*
10.0.0.0/32	10.10.0.0/24

Figure 2.5: Example match table demonstrating the overhead of BDDs.

2.1.1 Challenge: Expressiveness and Efficiency

It has generally been shown [6, 8] that using PECs improves the efficiency of the analysis. However, the specific data structure used to encode and reason about rule match conditions as well as the algorithm to perform the partitioning have significant effects on the resulting analysis tool. These factors impact (1) *Expressiveness*: the type of match conditions that can be encoded and analyzed by the tool (2) *Efficiency*: the ability of the tool to generate the PECs for large networks with a small time and memory footprint. In addition, the number of generated PECs can impact the efficiency of the subsequent PEC-based analysis.

As shown in Figure 2.4 the PEC-based analysis tools in existence today focus only on one of the two aspects. For example, Atomic Predicates Verifier [8] (APV), one of the first PEC-based network verification tools, uses Binary Decision Diagrams [27] (BDDs) to encode rule match conditions and to compute the PECs. BDDs are flexible DAG-based data structures that can compactly represent Boolean functions. In this setting, each Boolean variable corresponds to one bit in the packet header, and a Boolean function corresponds to the set of packet headers for which the function evaluates to true.

BDDs can encode any Boolean function and support logical operations such as conjunction, disjunction, negation, and overwrite over these functions. So BDD is a suitable choice of data structure for encoding many types of match-action tables that are used in practice today.[2].

[2]Although BDDs are not readily suitable for string-based match conditions such as URL matches in Web Application Firewalls [28]

However, BDDs are known to incur significant performance overhead per each bit in the packet header. For instance, consider the single match condition shown in Figure 2.5. To encode this single match condition using a BDD, we would need at least $(32 + 24)$ times the overhead of each node in the BDD's graph (per each non-wildcard bit in the match condition). Each BDD node would at least require three machine words worth of data (pointers to the left and write children, index of the variable the node corresponds to, etc.). So encoding that match condition would require more than 160 machine words. The performance of operations such as conjunctions is also proportional to the number of required machine words. The significant overhead per packet header bit has made APV a performance bottleneck in analyzing large real-world networks [14].

This has prompted recent PEC-based tools to use more efficient representations. For example ddNF [14], uses Ternary Bit Vectors (TBVs) to encode match conditions. TBVs are bit vectors with arbitrary wildcard bits. E.g., TBV $1x$ represents the set $\{11, 10\}$. Each ternary bit would take two bits to encode, therefore the TBV encoding of the table in Figure 2.5 would take only 2 machine words (assuming 64-bit words). The intersection of such match conditions would take as low as 3 ALU operations for the table. As a result, ddNF is nearly an order of magnitude more efficient than APV [14]. Similarly, VeriFlow [6] and Delta-net [5] use ranges to encode match conditions, and achieve better performance than APV.

However, the restricted encoding used in the more efficient approaches comes with a cost. None of these approaches can (efficiently) encode rules such as the ones used in our example (Figure 2.1), which are commonplace in networking (e.g., in Linux iptables rules). This is because of the existence of fields containing certain match types such as ranges (e.g., ports 1-1000), or sets of values, including complemented sets. Such values usually cannot be represented by a single, say, TBV [3]. Consequently, the corresponding tools cannot be used to (efficiently) analyze the more complicated types of match conditions.

A practical network analysis tool must be both efficient and expressive to handle the large and complex networks that are deployed in the real world. So the question is can we have a framework that achieves both? In this chapter, we answer this question positively by providing a new PEC-based analysis framework, called #PEC that resolves the tension between efficiency and expressiveness. #PEC can efficiently encode and reason about match types that ddNF, VeriFlow, or Delta-net can not encode. In addition, our rigorous experiments with real-world data-sets demonstrate that #PEC is at least 10× faster than APV in constructing PECs.

[3] Unless we break each match condition containing such values into multiple match conditions, each fitting into a single TBV, which can lead to exponential blowup in the number of match conditions.

Rule	Match	Action
	Destination	
1	10.0.0.0/17	Drop
2	10.0.128.0/17	Receive
3	10.0.0.0/16	Forward

Figure 2.6: Table of d_3, an example to demonstrate the problem with empty PECs

Match	*Action*
PEC_x	Drop
PEC_y	Receive
PEC_z	Forward

Figure 2.7: PEC-based equivalent of d_3.

2.1.2 Challenge: Empty PECs

There is a subtle challenge with certain PEC construction schemes. Specifically, #PEC and ddNF may produce empty PECs. If not detected and eliminated, analysis based on the resulting PECs may yield wrong results, including both false negatives and false positives.

Consider the table (d_3) in Figure 2.6. Let us assume the table in connected to another device (d_4) that directs all traffic back to d_3. Consider the following definition of PECs: $PEC_x = \{$packets in 10.0.0.0/17$\}$, $PEC_y = \{$packets in 10.0.128.0/17$\}$, and $PEC_z = \{$packets in 10.0.0.0/16 not in PEC_x or $PEC_y\}$.

This definition is similar to how ddNF and #PEC would define PECs (described in § 2.2.2). The definition also satisfies the conditions mentioned in § 2.1 for PEC-based analysis.

Figure 2.7 shows the PEC-based equivalent of rules in d_3 according to the PEC definitions above. Packets in PEC_z get forwarded to d_4 and sent back to d_3. So a naive analysis based on the defined PECs would report a forwarding loop. However, note that PEC_z is empty. Therefore, in reality, no packet would experience a forwarding loop in the setup of Figure 2.6. I.e., the reported loop is a *false positive*.

Moreover, it is common to analyze network tables to find *dead* forwarding rules: rules that are overshadowed by higher priority rules and can not possibly match on any packet. Say, one is interested in performing such an analysis for the table in Figure 2.6. A naive analysis based on the PECs defined above would fail to detect that rule 3 is a dead rule, a *false negative*. The reason is that the analysis would find PEC_z to be affected by that rule. In reality, since that PEC is empty, no packet is affected.

These examples underline the importance of detecting and eliminating empty PECs. Our case studies with real-world datasets also demonstrate the existence of false positives and

false negatives in tools that leave empty PECs unchecked, namely ddNF.

Detecting that PEC_z in the example above is easy. In general, however, doing so requires reasoning about the multi-dimensional packet header space, a coNP-hard problem [29]. We show that naive logical solutions based on SAT/SMT solvers or BDDs make PEC emptiness checking a performance bottleneck (§ 3.3.7). #PEC employs a packet-counting-based method (§ 2.2.3) to this problem that outperforms the naive solutions by $10 - 100\times$, effectively eliminating the bottleneck. In addition, in § 2.2.4 we show that by removing the empty PECs, #PEC achieves minimality in the number of generated PECs .

2.2 #PEC FRAMEWORK

This section overviews our insights and the technical approach for overcoming the challenges mentioned in the previous section. We provide a PEC-based packet header analysis framework that achieves both expressiveness and efficiency and does so without sacrificing the soundness or completeness of the analysis. #PEC achieves expressiveness by providing an extensible library of various efficient *match types* (§ 2.2.1). It achieves efficiency by using a lattice-theoretical approach to define and compute PECs using only intersection and subset operation, refraining it from using expensive negation/subtraction operations (§ 2.2.2). It also achieves precision (lack of false positives/negatives) by providing an efficient approach for detecting empty PECs based on model-counting that is orders of magnitude faster than baseline approaches (§ 2.2.3). This also enables #PEC to produce the minimal number of PECs (§ 2.2.4).

2.2.1 Flexible Match Types

Instead of bounding itself to a particular encoding, #PEC provides a library of efficient types (called match types), that the user can mix and match to encode the match conditions of the rules in match-action tables. The library includes TBVs, ranges, and other types that can not be efficiently encoded with TBVs or ranges, including (possibly complemented) sets. Table 2.1 illustrates some of the efficient match types in our library. The user can encode match conditions of tables using a tuple of various types (`tuple` is a match type itself). For example, the match conditions of Figure 2.1 can be encoded by the match type `tuple<ip_prefix, set<bv<8>>`.

The PEC computation algorithm of #PEC imposes a few restrictions on the match types: each type must provide efficient equality (=) and intersection (∩) operations. In addition, the number of concrete headers a value of the type represents (its `cardinality`)

Match type	*Description*	*Example values*
ip_prefix	IP prefix (v4 or v6)	10.0.0.0/24
exact<T>	Wildcard or a value of type T	TCP, Any
bv<N>	Fixed-length bitvector	1001101
tbv<N>	Fixed-length TBV	10xx0x1
range	Half-closed interval	[0 : 100)
disjoint_ranges	List of disjoint ranges	{[0,10), [20, 30)}
set<T>	Finite (possibly complemented) value set	{TCP, UDP}, !TCP
tuple<$E_1, \ldots, E_k$>	Tuple where E_j are element types	<10.0.0.0/24, !UDP>

Table 2.1: Part of #PEC's extensible match types library.

must be finite and efficiently computable. #PEC also depends on the subset ($\subset$) operation, but it can be computed indirectly using intersection and equality operations. In practice, a match type may provide a direct implementation of subset for better performance. Note that all types in our library satisfy all these requirements. For example, the equality, intersection, and subset of type ip_prefix are trivially defined. In addition, the cardinality of a prefix is the number of IP addresses represented by it. E.g., cardinality(10.0.0.0/24) = 256. For the type tuple, the intersection is the point-wise intersection of elements; the equality and subset are the conjunction of point-wise equality and subset of each element, respectively; and the cardinality is the product of the cardinality of each element. E.g., <10.0.0.0/17,!UDP>∩<10.0.0.0/16, {TCP,UDP}> = <10.0.0.0/17, {TCP}>, and cardinality(<10.0.0.0/24, {TCP,UDP}>) = 512. We provide efficient implementations of the match types in #PEC's library. The details can be found in [29].

The library can also be extended with any other type that satisfies the requirements mentioned above. For example, one could extend the library with a restricted form of regular expressions on bounded length strings such as URLs. This way, #PEC can reason about, say, Web Application Firewalls [28] rules that even APV cannot encode and analyze.

By decoupling the PEC construction algorithm from the encoding of match conditions and supporting any type supporting a few basic operations, #PEC strictly generalizes the expressiveness of other tools.

2.2.2 Implicit Subtractions

One of the main insights of this work is the observation that computing the result of subset and intersection operations on our match types such as TBV, ranges, and prefixes are often significantly more efficient than computing the result of negation/subtraction operations. For example, computing the intersection of two 8-bit TBVs (tbv<8>) would

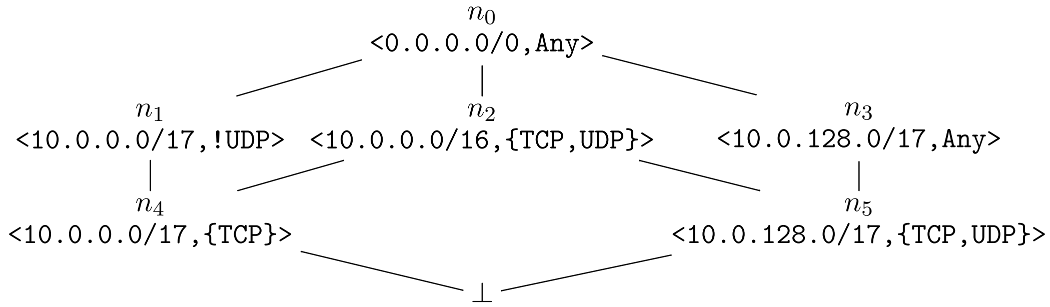


Figure 2.8: The SICL for the running example in Figure 2.1.

require as low as only 2 ALU operations, and the result can be represented by a single TBV fitting into a single machine word. However, merely representing the result of a single subtraction of TBVs such as $xxxxxxx - 000000$ would require at least 8 TBVs: $\{1xxxxxxx, x1xxxxxx, xx1xxxxx, xxx1xxxx, xxxx1xxx, xxxxx1xx, xxxxxx1x, xxxxxxx1\}$. Note that this is for a single subtraction of only 8 bits of TBVs (corresponding to, say, a single field in the packet header). Analyzing the tables of real networks require many more subtractions/negations over many packet header fields because one has to subtract the match conditions of higher priority rules from the ones in the lower priority rules to determine the packets that are affected by the lower priority rules (§ 2.1).

Our idea is to define and compute our PECs without explicitly evaluating the result of subtraction/negation operations. To do so, after encoding the match conditions of tables with a user-defined match type, we use an efficient algorithm (see below) to compute the closure of all match conditions under the intersection operation and organize the results in form of a Set Intersection-Closed Lattice [30] (SICL). Essentially, we organize the values in a DAG where each node represents a (unique) match condition or the intersection of two or more conditions and there is an edge from x to y iff y is a maximal subset of x (i.e., $y \subset x$ and there is no superset of y that is a subset of x). This DAG is also referred to as the Hasse diagram of the corresponding lattice[4].

For instance, Figure 2.8 represents the generated SICL for our running example in Figure 2.1. n_1 and n_2 correspond to the match conditions of rule 1 and 2 in d_1, respectively, and n_3 corresponds to rule 1 in d_2. n_0 is the universal set for the match type (also corresponds to rule 3 in d_1 and rule 2 in d_2). n_4 is the intersection of the match conditions in n_1 and n_2 and n_5 is the intersection of the match conditions in n_2 and n_3. $\perp$ is the empty set and will be ignored in our discussions.

[4]Throughout the chapter we refer to a SICL and its DAG/Hasse diagram representations interchangeably.

Note that the construction of the DAG mentioned above only requires supporting the intersection, subset, and equality operations. Moreover, the structure of the DAG allows us to define our PECs in a way that that does not require computing any negation/subtractions explicitly, yet we can be sure that the resulting PECs satisfy the requirements of PEC-based analysis.

Defining PECs based on SICL Recall from § 2.1 that the PECs used for formal network analysis should satisfy two basic requirements: (1) The PECs must be disjoint from each other, and (2) all packets belonging to the same PEC must experience the same forwarding behavior throughout the network.

We use the DAG structure of the constructed SICL of match conditions as follows. We define one PEC per node in our DAG. For any node n, let *n.match* be the value associated with that node in the lattice (a match condition of a rule or the intersection of multiple match conditions); let *n.children* be the direct children of n; and let *n.descendants* to be all the descendants of n in the DAG. We define the PEC associated with node n (PEC_n) to be the set consisting of all packet headers represented by *n.match* except the ones that are represented by the PECs defined by *n.descendants*:

$$Packets(PEC_n) = n.match - \bigcup \{Packets(PEC_d) : d \in n.descendants\}$$

. It is easy to show that the definition above is equivalent to

$$Packets(PEC_n) = n.match - \bigcup \{c.match : c \in n.children\}$$

For example, PEC_{n_2} represents packets that are matched by `<10.0.0.0/16,{TCP,UDP}>` but are NOT matched by `<10.0.0.0/17,{TCP}>` or `<10.0.128.0/17,{TCP,UDP}>`, essentially UDP packets with destination IP in 10.0.0.0/17 (regions 3 in Figure 2.2).

Note that the formulae above define the *semantics* of each PEC (the set of packets represented by a PEC). One does not need to evaluate the formulae for PEC-based analysis. The PECs are simply labels that can be used during the analysis. The section below describes how one can turn the match conditions or logical queries into sets of PECS. This way, the user will only deal with sets of labels (PECs) rather than logical constraints. The only important point is whether our definition of PECs satisfies the requirements of PEC-based network analysis mentioned above. In § 2.2.4, we prove that it does.

With this approach, we refrain from using expensive logical negation/subtraction operations. The negation is implicitly encoded in our definition of PECs based on our lattice structure

Algorithm 2.1: Converting logical queries to set operations over PECs

```
1  function Convert_to_PEC(query):
2      if query is a match value then
3          n ← Find_Node(query);
4          return Subtree(n);
5      else if ∃ g : query = ¬g then
6          Universe ← Subtree(Root);
7          return Universe − Convert_to_PEC(g);
8      else if ∃ g, h : query = g ∧ h then
9          G ← Convert_to_PEC(g);
10         H ← Convert_to_PEC(h);
11         return G ∩ H;
12     else if ∃ g, h : query = g ∨ h then
13         G ← Convert_to_PEC(g);
14         H ← Convert_to_PEC(h);
15         return G ∪ H;
```

but is never evaluated.

PEC-based network analysis As shown in § 2.1, once the PECs are defined, the formal network analysis can be based entirely on PECs. Abstractly, we show how to resolve any logical query about packet headers using the constructed PECs in a network consisting of tables for which we create the PECs. Intuitively, any logical query about packets consisting of propositions about packet headers and Boolean operations can be turned into equivalent sets of PECs and set operations. This process is described in Algorithm 2.1. The function performs a structural recursion on the input query and returns the set of all PECs for the packets of which the query evaluates to true. Particularly, predicates are converted into sets of PECs, and negation, conjunction, and disjunction are converted into set complement, intersection, and union, respectively. We show an example of this conversion in § 2.3.3.

The algorithm assumes that the logical query is a Boolean combination of logical predicates that have the same match type as the nodes of the constructed SICL, and the predicates can be found in the lattice. If a predicate does not exist, one can simply insert it into the SICL before running the query. The correctness of the algorithm can easily be derived from our definition of PECs in this section.

SICL construction algorithm We use an incremental algorithm for building the SICL from the match conditions of forwarding rules. Initially, the lattice only contains the match value corresponding to the universal set of the desired match type (T) – e.g., n_0 in Figure 2.8.

Algorithm 2.2: Insert a match value into SICL and update its structure

```
1  function Insert(match):
2  |   n, new ← Find_Or_Create_Node(elem) ;
3  |   if new then
4  |   |   Modified_Nodes.insert(n) ;
5  |   |   Insert_Node(Root, n) ;
6  function Insert_Node(parent, n):
7  |   Γ ← {} ;
8  |   for child ∈ parent.children do
9  |   |   if child.elem ⊆ n.elem then
10 |   |   |   Γ.insert(child) ;
11 |   |   else if n.elem ⊆ child.elem then
12 |   |   |   Insert_Node(child, n) ;
13 |   |   |   return;
14 |   |   else
15 |   |   |   e′ ← n.elem ∩ child.elem ;
16 |   |   |   if e′ is not empty then
17 |   |   |   |   n′, new ← Find_Or_Create_Node(e’) ;
18 |   |   |   |   Γ.insert(n′) ;
19 |   |   |   |   if new then
20 |   |   |   |   |   Modified_Nodes.insert(n′);
21 |   |   |   |   |   Insert_Node(child, n’) ;
22 |   parent.children.insert(n);
23 |   Modified_Nodes.insert(parent);
24 |   max_children ← {c ∈ Γ | ∀c′ ∈ Γ: (c.elem ⊆ c′.elem → c = c′)} ;
25 |   for max_child ∈ max_children do
26 |   |   parent.children.erase(max_child);
27 |   |   n.children.insert(max_child) ;
```

In each iteration, a new match condition (encoded in match type `T`), is inserted into the data structure, and the lattice is incrementally updated to keep the set closed under set intersection and to make the DAG structure retain the properties discussed above. After inserting all match conditions from all tables, we use the resulting SICL to define the PECs. The order of insertion of match conditions does not change the outcome as the DAGs constructed by different orders of insertions are isomorphic to each other.

Algorithm 2.2 details the insertion process. The user would call `Insert` with a match value. Initially, the procedure checks if the inserted match value already exists in the DAG, either due to insertion of the same match value before or due to the closure of previous match values under set intersection. If such a node does not exist, a new node is created and passed to `Insert_Node` that would place it in a proper place in the SICL. Under the

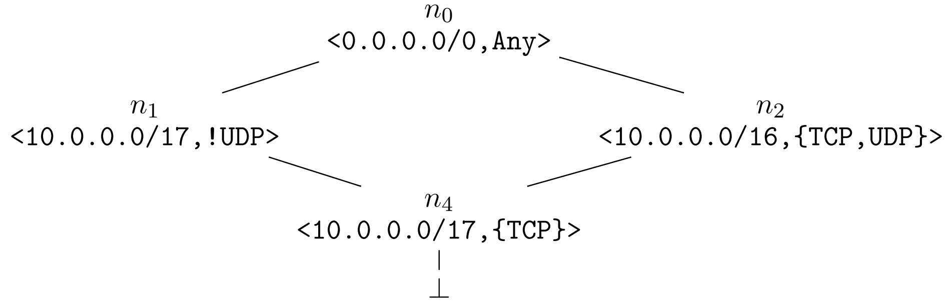


Figure 2.9: The SICL for match conditions in in Figure 2.1b.

hood, `Insert_Node(`r, n`)` recursively places n in the proper place in the subgraph rooted at r (initially the root of the DAG). The procedure analyzes the subset relation between n, r, and the children of r. If n is a subset of any children c of r, n is inserted recursively under c (Lines 12-13). Otherwise, n should be a child of r (Line 22). In that case, the non-empty intersections of n with r's children are computed and recursively placed in their proper places under the corresponding children (Line 21). Moreover, all subsets of n among *r.children* or their (non-empty) intersection with n are kept in Γ (Lines 10 and 18). For each maximal value m in Γ (Line 24), the lattice is updated to reflect the fact that m should be a child of n (Lines 26-27).

We illustrate the incremental insertion process with a simple example. Figure 2.9 shows the resulting SICL from inserting the match conditions of d_1 from our running example in Figure 2.1. Suppose we insert the remaining match condition from d_2. Calling `Insert(`*<10.0.128,0/17, Any>*`)` would create n_3 with that match value and call `Insert_Node` to place n_3 in a proper place under n_0. n_3 is not a subset of any of n_0's children so it must be a direct child of n_0. In addition, n_3 has a non-empty intersection with a child of n_0, namely n_2. Since the intersection is new, the function creates a new node (n_5) and insert it under n_2. It also adds n_5 to Γ. Since n_5 is the only node in Γ, is it a maximal children for n_3. Therefore the algorithm add it as a direct child of n_3. This results in the SICL updated to the one depicted in Figure 2.8.

We found Algorithm 2.2 empirically more efficient than the alternatives discussed in [14, 30]. The interested reader may refer to [30] for an overview of the alternative approaches and the proof of correctness of Algorithm 2.2. We also extend the original algorithm to keep track of the lattice nodes that are modified as a result of one or more calls to `Insert`. We use this information for detecting empty PECs (§ 2.2.3). In the example above, the set of modified nodes are $\{n_0, n_2, n_3, n_5\}$.

Algorithm 2.3: Compute/update the PEC packet count of modified nodes

```
1  function Compute_PacketCount(n):
2  |  queue ← [n] ;
3  |  visited ← {n} ;
4  |  n.packet_count ← cardinality(n.elem) ;
5  |  while queue is not empty do
6  |  |  n′ ← queue.dequeue() ;
7  |  |  for child ∈ n′.children do
8  |  |  |  if child ∉ visited then
9  |  |  |  |  if child ∈ Modified_Nodes then
10 |  |  |  |  |  Compute_PacketCount(child);
11 |  |  |  |  visited.insert(child) ;
12 |  |  |  |  n.packet_count ← n.packet_count − child.packet_count;
13 |  |  |  |  queue.enqueue(child) ;
14 |  Modified_Nodes.erase(n) ;
```

2.2.3 Detecting Empty PECs

#PEC represents negation/subtraction operations implicitly in the definition of PECs, without explicitly evaluating the operations. This may result in cases where a number of the defined PECs are empty.

In fact, ddNF, which shares the idea of implicit subtraction with our work, produces such empty PECs. Our experiments with real-world datasets (§ 2.3.3) show that analysis with ddNF can indeed lead to false positives in practice. Therefore, detecting empty PECs is necessary.

The general problem of PEC emptiness check is coNP-hard [29]. Nevertheless, the specifics of an empty PECs detection approach significantly affect its performance in practice.

A naive way of implementing PEC emptiness check is to check the satisfiability of the definition of a PEC by using BDD or SAT/SMT-based approaches. However, doing so would re-introduce the per-bit overhead, defeating the purpose of using more efficient encoding and our PEC construction approach in general. We experimentally demonstrate that using BDD or SAT-based solutions to the PEC-emptiness check problem is indeed inefficient § 2.3.4.

To mitigate this problem, our insight is that it would be enough to know the *number* of packets represented by a PEC to check whether or not it is empty. One does not need to explicitly compute the set of packets. Note that counting the number of packets in our PECs is not trivial either (it is a #P-hard problem [29]).

We provide an efficient algorithm that uses the already available structure of our SICL to count the number of packets in each PEC. The process is detailed in Algorithm 2.3. The

algorithm incrementally updates the packet counts of new/modified nodes since the last call to `Insert`. The set is produced by our SICL construction algorithm (Algorithm 2.2) and consumed by `Compute_PacketCount`. Intuitively, for each PEC_n (corresponding to SICL node n) whose packet count need to be computed or updated, the algorithm recursively computes/updates the packet count of the PECs among n's descendants (only if needed) and subtract the sum of these packet counts from $\texttt{cardinality}(n.match)$. The function can be called either after each update to the lattice, or once at the end, or anything in between, depending on the application and type of analysis (offline vs. real-time verification). After computing the packet counts of each PEC, one can simply look for PECs with the packet count of 0 to detect the empty ones.

For instance, in the running example in Figure 2.1, the packet count of n_5 is equal to the cardinality of $n_5.match$ ($= 2^{15} \times 2$) since the node does not have any descendants. Similarly $n_4.packet_count = \texttt{cardinality}(n_4.match) = 2^{15}$. n_2 has n_4 and n_5 as its descendants, so its packet count is equal to the cardinality of its match value minus the sum of packet counts of n_4 and n_5 (i.e., $2^{17} - (2^{15} + 2^{16}) = 2^{15}$). Similarly $n_1.packet_count = \texttt{cardinality}(n_1.match) - n_4.packet_count = 2^{15} \times 255 - 2^{15} = 2^{15} \times 254$ and $n_3.packet_count = 2^{23} - 2^{16}$. Finally, the packet count of n_0 is the cardinality of the universal set ($2^{32} \times 2^8$) minus ($\sum_{i=1}^{5} n_i.packet_count$). In this example, none of PECs are empty.

In the example corresponding to Figure 2.6, the resulting SICL would be a tree with three nodes beside the root node (0.0.0.0/0): 10.0.0.0/16 (node z) as the only child of the root and 10.0.0.0/17 (node x) and 10.0.128.0/17 (node y) as the children of z. The packet count of PEC_z is $2^{16} - (2^{15} + 2^{15}) = 0$, so the PEC is empty. In § 2.3.3, we overview more complex examples of empty PECs in real-world datasets.

Algorithm 2.3 is quadratic in the size of the DAG, which can be exponential in the number of input match conditions [14]. However, as shown by our experiments (§ 2.3.6), this process is fast in practice and outperforms the BDD and SAT/SMT-based solution by orders of magnitude.

2.2.4 Minimality of PECs

In this section, we show that the number of non-empty PECs produced by #PEC is minimal, in a certain sense of minimality defined by Yang and Lam [8]. More precisely, we show that the set of non-empty PECs produced by #PEC form *Atomic Predicates*.

Definition 2.1 (Atomic Predicates [8]). Let $\mathfrak{M}$ be a set of predicates, each representing a

match condition. Then atomic predicates of $\mathfrak{M}$ are the set of predicates $\mathsf{A}(\mathfrak{M}) = \{\alpha_1, \ldots, \alpha_k\}$, satisfying the following properties:

1. for all $i \in \{1, \ldots, k\}$, $\alpha_i \neq \textbf{false}$;
2. $(\bigvee_{i=1}^{k} \alpha_i) = \textbf{true}$;
3. $\alpha_i \wedge \alpha_j = \textbf{false}$ for all $i, j \in \{1, \ldots, k\}$ such that $i \neq j$;
4. Each predicate p in $\mathfrak{M}$, where $p \neq \textbf{false}$, is equal to the disjunction of some subset of atomic predicates:
$$p = \bigvee_{i \in S(p)} \alpha_i \text{ where } S(p) \subseteq \{1, \ldots, k\};$$
5. k is the minimal value of k such that the set $\{\alpha_1, \ldots, \alpha_k\}$ satisfies the above four conditions.

Yang and Lam show that the set of Atomic Predicates for a certain set of match-conditions is unique [8]. The authors provide a BDD-based approach for computing the Atomic Predicates from input match conditions. We show we can get the same through a fundamentally different (and more efficient) algorithm:

Theorem 2.1 (Minimality of #PEC). Given as input a set of match conditions $\mathfrak{M}$, the set of non-empty PECs constructed by #PEC forms atomic predicates $\mathsf{A}(\mathfrak{M})$.

Proof. The proof can be found in [29]. QED.

In addition, it is easy to show that a set of PECs being Atomic Predicates is a sufficient condition for it to be used for PEC-based analysis as described in § 2.1. However, it is not a necessary condition. In fact, among the PEC-based analysis tools mentioned in this chapter, only APV and #PEC produce Atomic Predicates. In addition, as demonstrated in the next section, #PEC does so an order of magnitude faster than APV (§ 2.3.5).

2.3 EVALUATION

We experimentally evaluate #PEC and compare it with other PEC-based network verification tools (namely APV, ddNF, and VeriFlow) in terms of performance and expressiveness. In doing so, we use various real-world datasets including ones from campus networks, data centers, and real-world firewall configurations,

Using ddNF, we also experimentally demonstrate that empty PECs can lead to wrong analysis results while analyzing real-world networks, a problem that #PEC efficiently avoids.

We compare the performance of #PEC's counting-based PEC-emptiness checking approach with the baseline approaches based on SAT/SMT and BDD solvers.

Overall, our experiments show that #PEC achieves both expressiveness and efficiency, making it a suitable framework for practical network verification.

2.3.1 Implementation

Here we outline our implementation of #PEC and the related frameworks, as well as the implementation of PEC-emptiness checking approaches.

Implementation of APV, ddNF and #PEC To rigorously evaluate the performance of our tool against others, we implement a version of APV and #PEC within the same framework: we opted for Z3 [35]. Our re-implementation of APV applies the same optimizations as proposed in [14]. We do not have to re-implement ddNF, since it is already available as an open-source module in Z3. Similar to ddNF, our implementation of #PEC leverages Z3's highly optimized TBV implementation. We implement the other element types as a C++11 library, which we describe in more detail in [29].

PEC-emptiness checking approaches In addition to implementing #PEC's counting method, we want to evaluate the SAT/SMT and BDD-based solutions to the PEC-emptiness problem that use propositional logic to precisely encode when a PEC is empty. Their symbolic encoding works as follows.

Let n be a node in our SICL. To check the emptiness of $Packets(PEC_n)$, we encode the definition in § 2.2 as the propositional logic formula $n.match \wedge \neg \left(\bigvee_{c \in n.children} c.match\right)$. For checking the formula's satisfiability, we use an SAT/SMT solver or construct a BDD, as detailed next.

Our BDD implementation uses the C++ BuDDy library. We set the initial node number

Dataset	*Summary*
REANNZ-IP [31, 32]	1,159 distinct IP prefixes
REANNZ-Full [31, 32]	1,170 OpenFlow rules
Azure-DC [33]	2,942 ternary 128-bit vectors
Berkeley-IP [5, 34]	584,944 distinct IP prefixes
Stanford-IP [7]	197,828 distinct IP prefixes
Stanford-Full [7]	2,732 ternary 128-bit vectors
Diekmann [31]	Thousands of 8-tuples

Figure 2.10: Summary of datasets

and cache size by manual tuning and choosing values that yield better results. In the case of the SAT/SMT implementation, we call Z3 [35]. To avoid additional parsing overhead, we use Z3's C++ API to construct the Boolean formulas, rather than using the more standard SMT-LIB [36] format for SAT solvers.

As part of the Boolean encoding of match types (recall § 2.2.1), we convert `tbv<N>` elements into `N` Boolean variables, one for each non-wildcard ternary bit. For the conversion of `set<T>`, which is implemented using bitsets, we encode the disjunction of the indexes of the set bits using $\lceil \log_2 K \rceil$ Boolean variables where K is the length of the bitset. For the `tuple<E`$_1, \ldots,$`E`$_k$`>` encoding, we designate $b = b_1 + \ldots + b_k$ Boolean variables where b_j is the number of Boolean variables needed to represent $\mathtt{E}_j$. The final encoding is the conjunction of the Boolean encoding of each tuple coordinate.

2.3.2 Datasets

To rigorously evaluate #PEC, we use 64 different datasets extracted from five independent routing tables and firewall collections [5, 7, 31, 37]. Figure 2.10 summarizes the datasets. Each dataset is encoded as a list of rule match conditions of a suitable match type (§ 2.2.1). Since ddNF only supports TBVs, we encode the match conditions in our datasets as TBVs whenever possible. This is not always possible though, particularly in the 'Diekmann' dataset, as described below. We pre-process the match conditions to eliminate duplicates. We describe each category of datasets in turn.

REANNZ: The REANNZ-Full dataset [37] contains more than a thousand OpenFlow rules, extracted from a single routing table that was used in the Cardigan deployment [32]. The OpenFlow rules in the REANNZ dataset use the following header fields: source and destination MAC addresses, ether-type, source and destination IP addresses, IP protocol field, and source and destination TCP ports. We convert each match condition in the rules to a 216 bit TBV. From the full dataset, we extract REANNZ-IP which contains only IP prefixes, but also encoded as TBVs.

Berkeley-IP: The Berkeley-IP dataset originates from [5] where IPv4 prefixes from the RouteViews project [34] were evaluated in the context of the UC Berkeley campus network topology. Our dataset focuses only on the IPv4 prefixes, which we encode as 32-bit long TBVs.

Azure-DC: The Azure-DC dataset [33] contains FIBs that simulate Azure-like data centers as deployed by Microsoft at that time. It contains a total of nearly 3000 match conditions, each of which is a 128-bit TBV.

Stanford: The Stanford dataset originates from Stanford's backbone network [7], which

contains configurations of sixteen Cisco routers. For each router, we generate its transfer function [7] which models the static behavior of the router (including forwarding and ACLs). We then use the match conditions in the transfer function, encoded as 128-bit TBVs, to produce a dataset for that router (e.g Stanford-Full/boza). To measure the effect of analyzing a network containing all sixteen routers, we also combine all sixteen datasets into a single one, Stanford-Full, which contains a total of 2,732 unique ternary 128-bit vectors. In our Stanford-IP dataset, we extract the IP prefixes directly from the raw router configurations, thereby avoiding the IP prefix compression feature in HSA's transfer functions. As a result, our Stanford-IP datasets are significantly larger than the datasets used in the evaluation of HSA [7] and ddNF [14].

Diekmann: The Diekmann datasets contain match conditions from real-world Linux iptables rule-sets [31]. We parse the following packet header matching fields: source and destination IP prefix, source and destination port, protocol, connection state, input, and output interface. We encode these as a mixture of TBVs and regular bitsets, which we combine into 8-tuples. We ignore wildcard characters for interfaces. Since iptables does not strictly follow the match-action abstraction introduced in § 2.1, we simplify each original iptables rule-set through a pre-processor that propagates match conditions along iptables chains in a depth-first manner. This essentially flattens a multi-chain iptables configuration into a list of match conditions without jumps and returns, conforming to our assumptions.

2.3.3 Case Study: Empty PECs

To emphasize the importance of detecting empty PECs, we describe real-world cases of imprecision in ddNF, all of which #PEC handles successfully. Due to space, we only illustrate a few examples (in our full study, we encountered over three dozen cases of imprecision in ddNF).

False negatives: Due to the existence of empty PECs, ddNF misses 35 dead rules (§ 2.1.2) in the REANNZ dataset. We also identify four dead rules in the Stanford datasets that are missed by ddNF, one in each of the 'soza', 'sozb', 'yoza', and 'yozb' Cisco routers.

False positives: In the Stanford dataset, we identified a case of false positive in answering the query: *"Would every packet with the destination IP address in `171.64.79.160/24` be forwarded from router 'yozb' to router 'yoza"?*. For this query, ddNF wrongly reports that some packets with such a destination IP address are dropped. The relevant rules of the 'yozb' router are shown in Figure 2.11 (slightly simplified for representation).

Here, ddNF produces this wrong result, because the union of IP prefixes that forward to 'yoza' equals the IP prefix of the last rule that drops packets: the match condition of the last

```
Destination=171.64.79.160/28 => yoza
Destination=171.64.79.176/28 => yoza
Destination=171.64.79.128/27 => yoza
Destination=171.64.79.192/27 => yoza
Destination=171.64.79.224/27 => yoza
Destination=171.64.79.0/25   => yoza
Destination=171.64.79.0/24   => DROP
```

Figure 2.11: Part of 'yozb' router rules in the Stanford dataset.

```
Protocol=ICMP               => Controller
Destination=210.4.214.0/24 => Port 1
Destination=210.4.215.0/24 => Port 1
Destination=210.4.214.0/23 => Port 2
Destination=ANY             => DROP
```

Figure 2.12: Part of OpenFlow rules in REANNZ dataset.

rule, therefore, is encoded as a singleton set that contains an empty PEC — similar to the case described in § 2.1.2.

For a more complicated example, consider the OpenFlow rules in Figure 2.12 obtained from the REANNZ dataset (slightly simplified to help with readability), ordered from highest to lowest priority. Suppose we encode these rules with the match type `<ip_prefix, exact<bv<8>>`. The match values above induce the SICL illustrated in Figure 2.13. Say, a network operator wants to verify the following intent:

> "All packets destined to IP prefix `210.4.214.0/23`, except the `ICMP` packets, must be forwarded through Port 1."

To check the intent, the analysis need to check the fate of PECs corresponding to the formula `210.4.214.0/23,ANY` $\wedge \neg($`0.0.0.0/0,ICMP`$)$. The first and second propositions correspond to the match values of n_b and n_c in Figure 2.13, respectively. Algorithm 2.1 would turn this query to $Subtree(n_b) \cap (Subtree(n_a) - Subtree(n_c)) = \{PEC_{n_b}, PEC_{n_d}, PEC_{n_e}\}$. Figure 2.14 shows the fate of packets for each PEC according to the rules. PEC_{n_d} and PEC_{n_e} are indeed forwarded through Port 1, but PEC_{n_b} violates the intent. In such case, ddNF reports a violation of the intent. However, looking at the rules above, it is easy to see that the intent indeed holds. Therefore ddNF's report is a false positive.

To find the crux of this false positive, let us take a closer look at PEC_{n_b}. The packet counts of PEC_{n_g}, PEC_{n_h}, PEC_{n_d}, PEC_{n_e}, and PEC_{n_f} (the descendants of n_b) are $256, 256, 256 \times 255, 256 \times 255$, and 0, respectively (§ 2.2.3). The packet count of PEC_{n_b}, therefore, is

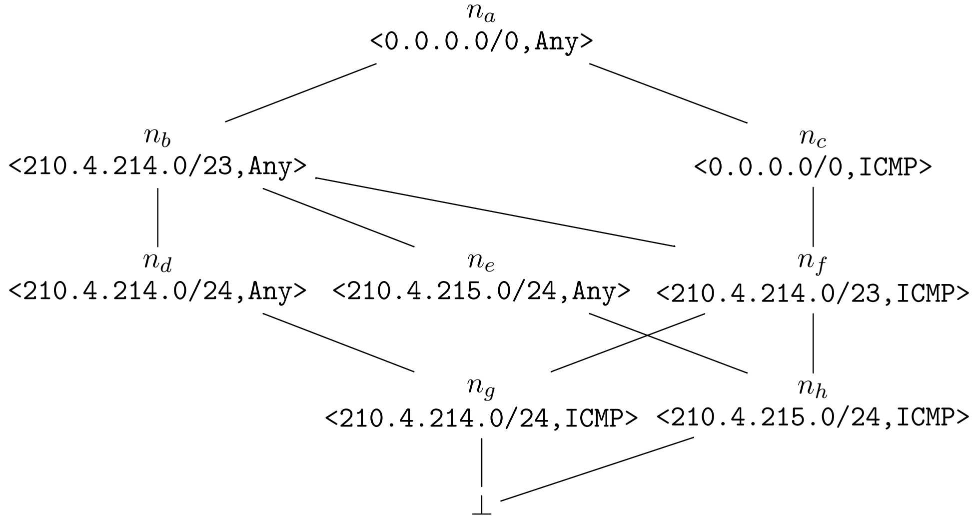


Figure 2.13: The SICL for part of REANNZ dataset.

Match	*Action*
PEC_{n_a}	Drop
PEC_{n_b}	Port 2
PEC_{n_c}	Controller
PEC_{n_d}	Port 1
PEC_{n_e}	Port 1
PEC_{n_f}	Controller
PEC_{n_g}	Controller
PEC_{n_h}	Controller

Figure 2.14: PEC-based equivalent of the rules in Figure 2.12.

$512 \times 256 - 256(1 + 1 + 255 + 255) = 0$. Thus, PEC_{n_b} is an empty PEC and the violation reported by ddNF does not affect any packet. #PEC correctly verifies that the property holds. For the sake of brevity, we omit the discussion of five other, but similar, examples of false positives in the REANNZ dataset.

Remark 2.1. In general, the source of false positives can be characterized as follows: let X and Y be two PECs where X is empty and Y is not, and let r_X and r_Y be two rules affecting X and Y, respectively. If r_X and r_Y have different actions, and the verification query includes both PECs, then ddNF's result is wrong when the property is true about X and false about Y.

Dataset	Insertions	PECs	Empty PECs	Atomic Preds.	PEC-construction time (s)		PEC-emptiness check (s)			APV (s)	Memory (MB)			
					Z3 ddNF	#PEC	BDD	SAT	Card.		BDD	SAT	Card.	APV
REANNZ-IP	1,159	1,160	25	1,135	<1ms	<1ms	0.016	0.414	<1ms	0.001	6	6	3	5
REANNZ-Full	1,170	12,783	275	12,508	0.112	0.009	2	9	0.018	3	14	26	9	10
Azure-DC	2,942	5,096,869	10,450	5,086,419	3301	121	20112	47829	30	25669	4,429	5,797	2,365	2,517
Berkeley-IP	584,944	584,945	29,813	Timeout	Timeout	2709	1553	460	0.515	Timeout	302	701	227	Timeout
Stanford-IP/soza	184,682	184,682	4,841	179,841	471	347	7	82	0.119	4951	102	251	69	49
Stanford-IP/yoza	4,746	4,746	3	4,743	<1ms	<1ms	0.076	2	0.002	2	8	9	4	6
Stanford-IP/All	197,828	197,828	4,874	192,954	266	199	19	89	0.156	5149	122	265	85	53
Stanford-Full/soza	524	16,764	81	16,683	0.056	<1ms	0.668	9	0.024	2	18	19	10	13
Stanford-Full/yoza	507	60,363	231	60,132	5	0.17	4	38	0.17	20	46	65	31	28
Stanford-Full/All	2,732	1,176,095	48,906	1,127,189	560	28	692	1958	4	2314	895	1,077	544	439
Diekmann/G	5,321	889,646	40	889,606	-	39	413	4729	10	2385	3,843	3,854	3,924	608
Diekmann/J	6,004	1,058,897	56	1,058,841	-	71	486	5654	13	2936	4,558	4,573	4,656	700
Diekmann/K	3,242	400,911	257	400,654	-	18	157	2084	3	732	1,997	2,006	2,031	233
Diekmann/P	578	492,378	4	492,374	-	47	168	1837	4	635	1,563	1,573	1,606	324
Diekmann/Q	307	4,626	38	4,588	-	0.087	0.763	17	0.016	0.94	21	29	18	7

Figure 2.15: Evaluation results for a subset of datasets. The full experimental results can be found in our technical report [29].

2.3.4 Performance Evaluation

We evaluate #PEC's performance along two dimensions, namely: (i) time and memory usage to construct #PEC's meet-semilattice; (ii) time and memory usage for detecting empty PECs. We discuss our results in turn.[5]

2.3.5 PEC Construction

We compare #PEC to APV, and Z3's implementation of ddNF. We ensure that every implementation benefits from the same optimizations (§ 2.3.1). We find that #PEC consistently outperforms APV and ddNF in Z3 where, on larger datasets, the speed-up is more than 10×. For example, on the Azure-DC dataset, our re-implementation of #PEC in Z3 is approximately 30× faster than ddNF. APV times out on the Berkeley-IP dataset after 10 hours, whereas #PEC completes the PEC construction in 45 minutes. We include in #PEC's total run-time including the time it takes to check PEC emptiness when comparing #PEC and APV. For this comparison, we use the 39 datasets in which either APV or #PEC runs for more than 100 ms, excluding the Berkeley-IP dataset where APV times out. In 95% of these 39 cases, despite #PEC's PEC-emptiness check, #PEC is at least 10× faster than APV, and 25% of this time #PEC's speed-up is at least 100×. On average, #PEC is at least 80× faster than APV. Finally, APV and #PEC's memory usage averages out to be the same across these datasets. Figure 2.15 shows parts our experimental results, see [29] for the full details.

[5] All experiments are run on a Linux machine with an Intel Xenon CPU ES-1660 3.30GHz and 32GB DDR3 1333MHz RAM.

The fact that #PEC outperforms APV is expected since #PEC eliminates the per-bit overhead of BDDs. The performance difference between #PEC and Z3's implementation of ddNF, in turn, can be explained in terms of the number of intersection and subset operations required to insert a new match condition into their respective data structure: their total run-time is proportional to these operations. For example, in the Stanford-Full dataset, #PEC requires 0.4 million whereas ddNF in Z3 takes 8 million such operations, a 20× improvement. #PEC's improvement over Z3's implementation of ddNF is similar on the other datasets.

2.3.6 PEC-emptiness Checks

We compare #PEC's counting method to the SAT/SMT and BDD-based solutions to checking PEC-emptiness. We evaluate the performance of PEC-emptiness checking using the 24 datasets in which #PEC runs for more than 100 ms. We perform the PEC-emptiness check after the PEC construction has been completed. We take extra precautions in our implementations to ensure a fair comparison (§ 2.3.1). Figure 2.15 shows that #PEC's counting method significantly outperforms the SAT/SMT and BDD-based approaches: #PEC achieves at least a 10× speed-up compared to the SAT/SMT and BDD-based approach in over 95% of cases. On average, #PEC is at least 500× and 200× faster than the SAT/SMT and BDD-based approaches, respectively.

To understand why #PEC's counting-based approach outperforms the SAT/SMT and the BDD-based approaches, reconsider the IP prefixes in § 2.1.2. Representing x, y, and z in propositional logic requires 16, 15, and 15 variable assignments respectively, corresponding to their non-wildcard bits. Just encoding $z.match - (x.match \cup y.match)$ in SAT requires near 50 logic gates, excluding the task of checking satisfiability. Representing the predicates using BDDs requires the same number of BDD nodes. Assuming logical BDD operations are linear in their operand size, computing $Packets(PEC_z)$ at least requires CPU cycles proportional to the cumulative size of the three BDDs. On the other hand, the cardinality of each predicate in the example fits into a single machine word. We need only 2 arithmetic CPU operations to compute the packet count of PEC_z (i.e., $|z| - |x| - |y|$), and then check if it is zero. While in theory there are still near 50 operations performed (at the bit level), #PEC harnesses the computing power of ALUs to finish the operations in fewer CPU cycles. For example, in the Stanford-Full dataset where each node in the DAG has 3 children and 12 nodes in its subtree on average, the BDD-based approach requires 3×128 low-level BDD operations on average (each spanning tens of CPU instructions). By contrast, our counting-based approach needs at most 3 ALU operations for each subtraction. So #PEC

should be at least $(3 \times 128)/(12 \times 3) \approx 10\times$ faster than the BDD-based approach, and our experiments show indeed at least a $127\times$ speed-up.

2.3.7 Comparison with VeriFlow

We compare #PEC to the original implementation of VeriFlow [38]. Since that implementation of VeriFlow only supports a restricted form of OpenFlow rules where arbitrary per-field bitmasks are disallowed [39], it cannot analyze the majority of our datasets. We, therefore, restrict our experiments with VeriFlow to a simplified version of the Stanford-Full dataset. We use the default packet header field ordering. We ask VeriFlow to only find 'Equivalence Classes' (ECs), rather than each EC's forwarding graph. In this restricted setting, VeriFlow takes $41\,s$ to create 3,778,324 ECs, using $1\,GB$ of memory. Despite #PEC's support for arbitrary bitmasks, it is still more efficient than VeriFlow, in both time ($30\,s$) and space ($0.5\,GB$): specifically, #PEC constructs only 1,066,645 PECs in $27\,s$, and finds 44,418 empty PECs in 3s.

2.3.8 Discussion: Importance of Empty PECs

We showed that ddNF's wrong analysis results are due to the existence of PECs that are empty. In our case study (§ 2.3.3), we exemplified real-world cases where empty PECs lead to wrong analysis results. We emphasize that we only gave illustrative examples; our list is not exhaustive, and it includes cases where ddNF misses errors. In practice, therefore, ddNF is only as fast as the slowest decision procedure needed to sanity-check its results, a fundamental limitation. By contrast, #PEC's analysis is correct by construction, and its performance is *not* dependent on BDDs or SAT/SMT solvers, which are orders of magnitude slower in finding empty PECs (§ 2.3.6).

2.4 CONCLUSION

While PEC-based analysis improves the scalability of network verification, our experiments reveal the tension between expressiveness and efficiency in the PEC-base tools in existence. Our work, #PEC, offers a new PEC construction approach that resolves this tension. Expressiveness is achieved through using an extensible library of match types rather than bounding the PEC computation algorithm to a certain encoding. Efficiency is achieved by using the Set Intersection Closed Lattice of match conditions to define the PECs, eliminating the need for expensive subtraction operations. We also identified the problem of empty

PECs and our case study demonstrated that such PECs lead to imprecise analysis results in ddNF. We provided a counting-based solution to this problem that outperforms SAT/SMT and BDD-based solutions by $10 - 100\times$. In addition, #PEC constructs the unique minimal number of PECs, and it does so $10\times$ faster than APV's atomic predicates.

Chapter 3: Specification: Mining High-Level Intents from Low-Level Behavior [2]

There has been significant research progress towards network verification tools [1, 3, 5, 6, 10, 11, 12] that given a set of network-wide intents, check whether the configured network satisfies the intents. There has also been progress towards network programming/configuration synthesis [4, 40, 41, 42] tools that given the intents described in a domain-specific language, synthesize data plane entries or control plane configurations that satisfy the intents. These tools rely on the ability of an administrator to provide a formal (i.e. defined in a precise language) specification of the desired behavior. However, in practice such specifications are almost never available in a complete form. Intents typically begin as architectural design goals, and formalizing them in even a moderately large network would be an additional onerous burden for engineers. Moreover, administrators often inherit an already working legacy network without proper documentation and are asked to maintain the network. Therefore, administrators may not be fully aware of the how the network operates, even informally. That is part of why in the real world, administrators hesitate to touch the network they operate very often due to concerns over breaking the network [14].

The goal of this chapter is to mitigate the problem of unavailability of intent specifications. Our idea is to utilize the existing network behavior to infer the high-level intents. The inferred intents can subsequently be fed as input to verification and synthesis tools for continued maintenance of the network. If compact enough for human comprehension, the intents can also be viewed as a summary of network behavior which can assist the administrators in management and debugging. To take this approach, one needs to overcome challenges related to encoding behavior and intents, scalability, and imperfections in data collection.

We present Anime (*Automatic Network Intent Miner*), a framework and system to infer high-level intents by mining the common patterns among the low-level forwarding behavior in the network. We provide a setup to express both low-level behavior and the high-level intents via their associated features. Each feature corresponds to one aspect of the behavior, including packet header information, devices along the path, and environmental conditions such as time of path observation, device or link state, etc. We design our features to have hierarchical values (like in CIDR), providing fine-grained control over the trade-off between precision and recall. We then use this setup to formally define intent inference as an NP-hard constrained cost optimization problem related to our quality and compactness measures. We heuristically solve the problem by grouping relevant behavior using clustering techniques with the cost function as a measure of dissimilarity and finding the most specific intent that represents all behavior of each group. We also develop a suite of highly effective optimizations,

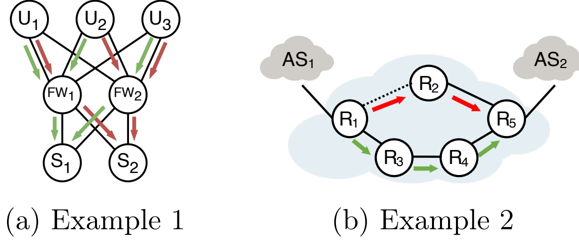
including indexing and parallelization, that allows our approach to scale to large networks with millions of forwarding paths.

Given observed forwarding behavior collected from one or more snapshots of the network, and a limit k on the number of inferred intents, Anime produces up to k intents that collectively describe all observations with high precision. The results also predict unobserved but possible behavior, which can be used to alleviate imperfect observations. The system can also be used to detect anomalous behavior.

We evaluate the effectiveness and performance of Anime on large synthetic and real-world operational networks with hundreds of routers and millions of forwarding paths. In our experiments we consider four use cases: (1) First, we consider a use case where the goal is to only summarize observed network behavior for human comprehension. (2) We also evaluate the tool in settings where not all possible behavior is observed and some needs to be predicted. (3) Next, we consider settings where the system is used to analyze multiple snapshots of network collected over time, with possibility of policy change over time. (4) Finally we use Anime to flag anomalous behavior and investigate its results. As baselines, we compare subsets of our results with the closely related work: Net2Text [43] which focuses on the summarization use case only, and dynamic invariant detection based tools such as Invar-net [44] and Config2Spec [45] which find properties that remain invariant over multiple snapshots. Our results show that Anime scales quasi-linearly with the number of inputs and can analyze large networks with an average runtime overhead of 3ms per input behavior. It can infer compact summaries (1000$\times$ smaller than input) with a small (1%) loss of precision. For imperfect observations, Anime can predict most of unobserved behavior with few false positives. In our multi-snapshot analysis, Anime can distinguish policy changes from noise. Moreover, Anime often generates significantly higher quality (higher F-score) results compared to Net2Text (as much as 70x in our experiments), and invariant based methods (up to 5x). Finally, our investigation of anomalous behavior flagged by Anime on a real-world operational network has led to interesting observations, including finding bugs in our data collection tools and in one case prompting actions by network operators.

3.1 MOTIVATION

A network operator needs to ensure that her network changes are safe. Network verification tools like [6, 46] can help in this regard. Such tools can readily be used to check generic properties (e.g. loop freedom). But for checking organization specific properties, such as reachability, isolation, or fault tolerance policies, these tools require formal specifications of correct behavior (intents), which are not available to the operators, and may not be even



(a) Example 1 (b) Example 2

Figure 3.1: Example network setups

known. The operator can manually investigate the network configurations or behavior to learn more, but it would be a burdensome task, with configurations often involving multiple people or teams, legacy configurations, and poorly-documented decisions by current and departed employees. Thus having a tool that automates or at least facilitates this process would be useful. This is where automatic intent inference comes into picture.

In this section, we first make the idea of intent inference more concrete via intuitive examples. We then discuss how inferred intents can be utilized by other systems.

3.1.1 Illustrative Examples

We provide two examples illustrating how forwarding behavior can be used to infer possible intents. The examples are inspired by the network verification literature [4, 5, 47]. In these examples, the information from forwarding paths across various devices, packet headers, and data plane snapshots are used to derive higher-level information about the collection of the paths.

Example 1: Data center network

Consider the network in Figure 3.1a resembling part of a very simple data center network. The network contains three user machines (U_1, U_2, U_2), two firewalls (FW_1, FW_2), and two servers (S_1, S_2). The green and red arrows in the figure denote the forwarding paths for packets with destination IP addresses 10.0.1.2 and 10.0.1.3 respectively.

Out of the three green paths destined to 10.0.1.2, consider two: $U_1.FW_1.S_1$ and $U_2.FW_1.S_1$. The only difference between these is their starting node (U_1 vs. U_2). Looking at these paths we could say that the enforced intent is that all packets starting from a user node destined to 10.0.1.2 go through firewall FW_1 and reach server S_1. We represent this guess as: $\{dstIP : 10.0.1.2, start : User, waypoint : FW_1, end : S_1\}$.

Let us now consider the third green path, namely $U_3.FW_2.S_1$, in addition to the previous

two. This path also starts from a user node and ends in S_1, but it goes through FW_2 instead of FW_1. Note that FW_2 is also a firewall. So we refine our initial guess of the intent to $\{dstIP : 10.0.1.2, start : User, waypoint : Firewall, end : S_1\}$, i.e. packets originating from the user nodes destined to 10.0.1.2 should traverse any firewall and reach server S_1. If we repeat this process for the paths destined to 10.0.1.3 (the red arrows), we get $\{dstIP : 10.0.1.3, start : User, waypoint : Firewall, end : S_2\}$.

Our two guesses for the green and red paths only differ in the last hop (S_1 vs. S_2). Both nodes are servers. So if we combine the information from all the paths, we can say that for packets destined to the prefix 10.0.1.2/31 originated at a user node, the packet will traverse a firewall and end up in a server node: $\{dstIP : 10.0.1.2/31, start : User, waypoint : Firewall, end : Server\}$.

Example 2: ISP network

Figure 3.1b illustrates another example network that resembles part of a very simple ISP. The routers in the blue cloud show the network of an Autonomous System (AS) under our administration connected to two other ASes, namely AS_1 and AS_2. Say we observe the behavior of packets destined to a specific IP prefix P for a period of one year (from January to December). Our observations indicate that from January to June, the packets destined to P received from AS_1 usually take the path shown by the red arrows in the figure: $AS_1.R_1.R_2.R_5.AS_2$. We also observe that when there is a link failure in that path (e.g. when $R_1 - R_2$ goes down), the packets destined to P take path indicated by the green arrows in the figure: $AS_1.R_1.R_3.R_4.R_5.AS_2$. By combining these observations obtained from multiple snapshots across time, we can say the common property held from January to June is that a packet destined to P received from AS_1 will hit R_1, traverse some internal nodes and reach AS_2 through R_5. This can be denoted as $I_1 = \{date : [Jan., June], failures : \{0, 1\}, dstIP : P, path : AS_1.R_1.Internal^+.R_5.AS_2\}$ where $^+$ denotes $\geq$1 repetitions.

Let's say starting from June, the operators decide to drop packets destined to P due to suspicious activities. So from June to December we observed that packets destined to P received from AS_1 get dropped at R_1. This can be represented by $I_2 = \{date : [June, Dec.], failures : Any, dstIP : P, path : AS_1.R_1.drop\}$. The set $\{I_1, I_2\}$ captures all observations from January to December.

3.1.2 Applications

Here we discuss some of the main applications of Anime.

Input for intent-based networking: The intents inferred from existing network behavior can be subsequently fed into verification or synthesis tools for continued maintenance of the network. More generally, the inferred intents can enable or streamline a suite of *Intent-Based Networking* applications including automatic migration from legacy networks to SDN or cloud paradigms, transparent network optimizations [48], automatic network repair [49], etc.

Network behavior summarization: When compact enough, the inferred intents can be viewed as a summary of network behavior that can assist network operators in understanding what is going on the in networks under their administration. This is particularly important as current network management relies heavily on humans in the control loop. Consequently, human insight is fundamental for network debugging [43] and management in general.

Behavioral anomaly analysis: The network from which the observed behavior is collected may include misconfigurations. Network verification tools are supposed help in detecting such misconfigurations, but in the absence of specifications of correctness, these tools cannot help. We can use Anime to detect behavior that are significantly dissimilar to the rest of the observed behavior (i.e. anomalies), and use that to detect possible network problems, not requiring correctness specifications. In this work we use Anime for detecting "negative anomalies" (behavior that should exist but does not). The framework can also be extended to detect "positive anomalies" (behavior that exists but should not).

3.2 ANIME FRAMEWORK

In this section we overview Anime's design. We first provide an abstract context for intent inference and use it to define our quality measures (3.2.1). We then describe our formal setup to express network behavior and intents via features with hierarchical values. We use the setup to formulate intent inference as an NP-hard constrained cost optimization problem (3.2.2). We provide a heuristic solution that groups similar behavior using clustering techniques and produces an intent that represents all behavior per each group (3.2.4). Finally, we describe optimizations that allow our technique to scale to large networks (3.2.4).

3.2.1 Problem Context

In our abstract view of the intent inference process (Fig. 3.2), there is a set of *actual intents* that govern the network behavior. Applied to the target network, the intents allow a set of *possible* forwarding behavior in the form of a set of forwarding paths in the network.[1] Note

[1] Here, we assume a white-listing model meaning that any path not explicitly allowed by any intent in a set of intents is disallowed by that set.

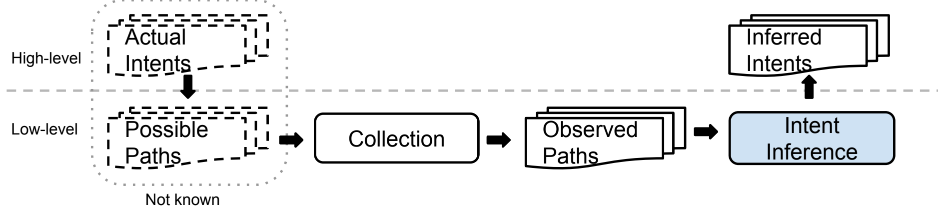


Figure 3.2: An abstract view of intent inference process

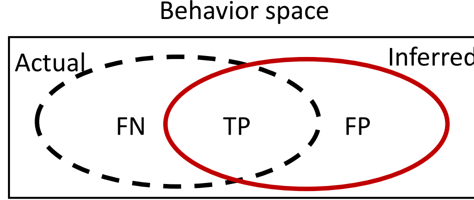


Figure 3.3: Quality setup

that the actual intents and the possible behavior are not known (otherwise we wouldn't need intent inference). Also for now, we assume that the actual intents are correct, i.e. the network is not misconfigured (we will revisit this assumption in anomaly analysis). A *collector* collects a subset of these paths by data plane or control plane configuration analysis [1, 5, 10, 12], monitoring actual network traffic, or any other means. The collection mechanism is orthogonal to our work but we emphasize that the collector may not be able to observe all possible paths. For example, a traffic-based collector may miss behavior not exercised by the traffic, or some possible behavior may only be visible during link failures (as in Example 2), etc. The *observed* paths are then fed into an *intent inference* tool which consequently generates a set of *inferred intents*.

We use this context to talk about quality of intent inference. The goal is to infer intents that are equal or close to the actual intents (ground truth). We can objectively measure the closeness by comparing the concrete behavior represented/allowed by the two sets of intents. Given the sets of actual (A) and inferred (I) intents, and function σ that maps a set of intents to the set of all concrete behavior represented/allowed by the intents, we define true positives as $TP \equiv |\sigma(I) \bigcap \sigma(A)|$, false positives as $FP \equiv |\sigma(I) - \sigma(A)|$, and false negatives as $FN \equiv |\sigma(A) - \sigma(P)|$. This is depicted in Figure 3.3,

This way we are able to objectively measure the quality through the classic notions of $Precision \equiv TP/(TP + FP)$ and $Recall \equiv TP/(TP + FN)$ which respectively correspond to the *specifity* (exclusion of incorrect behavior) and *coverage* (inclusion of correct behavior) of the inferred intents.

Our goal is to provide an intent inference process that achieves high precision and recall

but there is generally a trade-off between the two. In this work, we set to infer intents that represent all observed behavior with as much precision as possible. We note a special use case of intent inference, called summarization, where the set of possible paths (ground truth) is equal to the set of observed paths (perfect observation) and for human readability, the number of inferred intents is limited by a *compactness* parameter (k). Without the limit k, one can trivially list all observed paths as the inferred intents and achieve perfect precision and recall. We also consider and evaluate our tool in cases of imperfect observations. In that case, recall can be below 1 and k is interpreted as a hyperparameter to control model complexity thus avoiding over-fitting (low recall) and under-fitting (low precision).

3.2.2 Formal Setup

In Anime, both low-level forwarding behavior and the high-level intents are expressed through their associated features. Each *feature* captures one aspect of the forwarding behavior. Examples include packet header information (e.g. source/destination IP address, port, protocol), device information such as start and end points, waypoints, ingress, egress, entire forwarding paths, the observation timestamp, device or topology state (e.g link status), etc. In Example 1 we used a tuple of destination IP, start, waypoint, and end features. In Example 2 we used a tuple of time of observation, number of link failures, IP destination prefix, and entire forwarding path features.

Each feature can have a set of possible feature values or *labels* associated with it. One of the main insights of this work is to capture the hierarchies among feature labels that naturally fit networking environments. For example, 10.0.1.2 and 10.0.1.3 are both included in 10.0.1.2/31 which itself is a subset of, say, 10.0.1.0/24. Also in Example 1, both FW_1 and FW_2 are Firewalls. By supporting hierarchical values, Anime allows for finer grained precision-recall trade-off control, resulting in higher quality intents (Section 3.3).

To capture this, we define a *feature type* F as the tuple $((\Sigma_F, \subseteq), \delta_F)$ where Σ_F is the set of possible labels for F and $\delta_F : \Sigma_F \to \mathbb{R}$ is a cost associated with each label. We interpret the labels in Σ as labeled sets of values which are partially ordered by the subset ($\subseteq$) relation. F can essentially be represented by a DAG where nodes are labels in Σ_F and edges are (maximal) subset relation. Figures 3.4a and 3.4b show example feature types for the set of devices used in Examples 1 (D_{dc}) and 2 (D_{isp})[2], respectively. The number to the right of each label shows the cost of that label. For example $\delta_{D_{dc}}(Firewall) = 2$. Note how traversing from the bottom of these hierarchies to the top, the labels get less specific (lower precision)

[2]In Example 2 we assumed D_{isp} has a special node with label *drop* that is disjoint from the rest of the hierarchy.

but cover more values (higher recall). We assign a higher cost to higher loss of precision (see below).

Any label of Σ_F that is not a superset of any other label is called a *concrete* label of F, i.e. concrete labels are the leaves of the DAG representing F (denoted by σ_F). For label l, $\sigma_F(l)$ denotes the concrete labels under l (the subset of σ_F included in the set labeled by l) – e.g., $\sigma_{D_{dc}} = \{U_1, U_2, U_3, FW_1, FW_2, S_1, S_2\} = \sigma_{D_{dc}}(Any)$, $\sigma_{D_{dc}}(Server) = \{S_1, S_2\}$.

A *feature* is simply an instance of a feature type with a name.[3] We provide a library of features types (Section 3.2.3) that can be used to encode forwarding behavior. The library can easily be extended with additional feature types according to the definition above. The library includes a feature type for a tuple of multiple other feature types. We used the feature $F_{dc} = (dstIP, start, waypoint, end)$ as an instance of $Tuple\langle IPPrefix, D_{dc}, D_{dc}, D_{dc}\rangle$ in Example 1.

Within this setup, for a feature F, a *path* is simply a value from σ_F and an *intent* is a value from Σ_F. For instance $(10.0.1.2, U_1, FW_1, S_1)$ is a path and $(10.0.1.2, User, Firewall, S_1)$ is an intent of F_{dc}. We say an intent i represents a path p iff p is among the leaves under i, i.e. $p \in \sigma_F(i)$. For a set of intents $I = \{i_1, ..., i_k\}$ and a set of paths $P = \{p_1, ..., p_n\}$ we say I represents P iff for each path $p \in P$, there is at least one intent $i \in I$ that represents p, i.e. $P \subseteq \bigcup_{i\in I} \sigma_F(i)$. For instance $(10.0.1.2, User, Firewall, S_1)$ represents$\{(10.0.1.2, U_1, FW_1, S_1), (10.0.1.2, U_3, FW_2, S_1)\}$.

We define the intent inference problem in this framework:

Definition 3.1 (Intent inference problem)**.** For a given feature F, a set of paths P, and a limit on the number of inferred intents (k), find the set of intents $I^* = \{i_1, ..., i_{k'}\}$ ($k' \leq k$) (the *inferred intents*) that represents P and minimizes the sum of the cost of the intents, i.e. $\delta_F(I^*) = \sum_{i\in I^*} \delta_F(i)$.

For example for feature D_{dc}, $P = \{U_1, U_3, S_1\}$, for limits of 3, 2, and 1, it is easy to see the set of inferred intents are $\{U_1, U_3, S_1\}$, $\{User, S_1\}$, and $\{Any\}$ with the costs of 3, 4, and 7, respectively.

To understand the relation between this definition and the intuition provided in the last section, note that if we set the cost of each label to the number of concrete values it represents (as we mostly do in our feature library[4]), for any I in the set of all sets of intents representing P ($\mathbb{I}_P$), $\delta_F(I)$ approximates the number of concrete paths that the intents in I collectively represent, i.e. $TP + FP$[5]. Also, note that TP is the same for any such I. So $\delta_F(I)$ is inversely

[3]When it is clear from the context, we use feature and feature type interchangeably.

[4]See Sec. 3.2.3. One can also alter the costs to guide the inference.

[5]The imprecision is due to over-counting overlapping intents. We penalize overlap to encourage inferring disjoint intents.

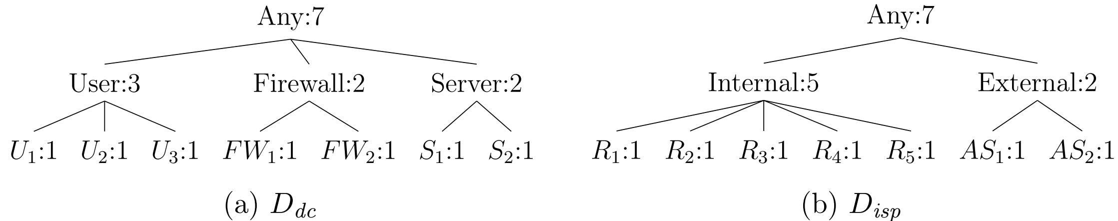


Figure 3.4: Device labeling hierarchy used in our examples

related to the precision of I and minimizing $\delta_F(I)$ maximizes the precision. Also for any such I, recall (on observed paths) is 1.

It is possible to show the problem defined above is NP-hard by a reduction from the set cover problem. We provide a heuristic polynomial solution to the problem in Section 3.2.4.

3.2.3 Feature Types Library

Inspired by [1], we provide a library of various feature types. Depending on the type of network, collection mechanism, intended application, domain-knowledge, etc, different type templates in the library can be instantiated to encode the forwarding behavior. One can also design additional feature types. The following are examples of templates supported in our library. Interested readers can refer to [50] for more types.

$DAG\langle V, E\rangle$ where V and E are nodes and edges of an arbitrary DAG, is a feature type where any label $v \in V$ is interpreted as the set of leaves reachable from v. D_{dc} is an example of this type.

Flat$\langle S\rangle$, where S is a set of concrete values, is a feature type defined by a DAG with a single root (*Any*) connected to $|S|$ leaves each corresponding to a member of S. An intent can either be a concrete value or *anything*, with no hierarchy in between. Net2Text [43] only supports this type of feature (Chapter 5).

$TBV\langle n\rangle$ is a feature defined over Ternary Bit Vectors (TBV) of length n. A TBV is a generalization of bit-vectors where arbitrary bits can be wildcards. A TBV is interpreted as the set of bit-vectors it represents, e.g. $0* \equiv \{00, 01\}$. *IPPrefix* is a specialization of $TBV\langle 32\rangle$ where wildcards can only happen at the end of TBVs.

Range. Integer ranges form a feature type suitable for fields like IP and port ranges or constraints like number of link failures.

$HRE\langle F, d\rangle$ is a variant of regular expressions over hierarchical alphabet useful for representing entire forwarding paths (Sec. 3.2.3).

Tuple$\langle F_1, ..., F_n\rangle$ combines the hierarchies of multiple features to create a more complex

feature. $\Sigma = \Sigma_{F_1} \times ... \times \Sigma_{F_n}$, and for any $a, b \in \Sigma$, $a \subseteq b$ iff $\bigwedge_{i\in[1,n]} a_i \subseteq b_i$. Finally $\delta(a) = \prod_{i\in[1,n]} \delta_{F_i}(a_i)$.

For all of these features (except HRE) $\forall l \in \Sigma : \delta(l) = |\sigma(l)|$. In other words, the cost of each label l is defined as the number of concrete label represented by l.

Representing entire forwarding path

As an exemplar of a complex feature, we describe a feature type designed to represent entire forwarding paths, such as the ones used in the ISP example.

Our idea is to use regular expressions to represents sets of such paths. Specifically, we focus on a limited class of regular expressions that seems to be a proper fit for representing network paths. The grammar of such regular expressions is shown in Figure 3.5. An HRE is defined over another feature type F and its alphabet is Σ_F. We slightly generalize the notion of acceptance in our regular expressions to account for the hierarchy of labels we introduced. We call these *Hierarchical Reduced Regular Expressions* (HRE) due to the restricted grammar and the extended notion of acceptance,

Here, a path is a string over the concrete labels of F (σ_F). We say a path p is represented by an HRE h iff there exists a string s obtained by replacing a subset of the labels in p with another label from the set of ancestors of that label in Σ_F and s is accepted by h interpreted as a normal regex. We denote by $Acc^d_F(h)$ the set of all strings over σ_F of length $\leq d$ represented by h. In the ISP example, the path $p = AS_1.R_1.R_3.R_4.R_5.AS_2$ is accepted by the HRE $h = AS_1.R_1.Internal^+.R_5.AS_2$ because $s = AS_1.R_1.Internal.Internal.R_5.AS_2$ which is obtained by replacing R_3 and R_4 in p by the label $Internal$ (ancestor to both device names in D_{isp}), is accepted by interpreting h as a normal regex. For a feature type F, and a limit d on length of strings, we define $HRE\langle F, d\rangle$ as $((\Sigma_H, \subseteq), \delta_H)$ where Σ_H is the set of all HREs over Σ_F with length $\leq d$. We interpret each $h \in \Sigma_H$ as the set $Acc^d_F(h)$ and the hierarchy is formed according to the subset relation among these sets, e.g. $h_0 = AS_1.R_1.R_2.R_5.AS_2 \subset h_1 = AS_1.R_1.Internal.R_5.AS_2 \subset h_2 = External.Internal^+.External \subset h_3 = Any^+$ ($d \geq 5$).

By the argument in Sec. 3.2.2 we should set $\delta_H(h) = |\sigma_H(h)| = |Acc^d_F(h)|$. However, computing $|Acc^d_F(h)|$ is expensive. Instead, we roughly approximate the value by $m_F(h)^d$ where $m_F(h)$ is the geometric mean of cost of labels of HRE h over field F (i.e. the average number of concrete labels of F each label in h represents). In Example 2, δ_H for $h_0, ..., h_3$ are 1^d, 1.38^d, 2.71^d, and 7^d, respectively. Note how less precise intents received higher costs.

In this dissertation, we only use HREs over a labeling hierarchy for network devices, such as D_{isp} in the ISP example. One can imagine HREs over more complex feature types such as tuples of the network device and packet header fields to track packet header changes along

$$\texttt{HRE} ::= l \mid l^+ \mid \texttt{HRE.HRE} \qquad l \in \Sigma_F$$

Figure 3.5: Grammar of Hierarchical Reduced Regular Expressions

the forwarding path.

3.2.4 Solving the Intent Inference Problem

Our heuristic solution to the intent inference problem divides it into two related sub-tasks: single intent inference and path selection.

Single intent inference

This is a specialization of the intent inference problem with $k = 1$: find a single intent that represents all input paths with the lowest cost. For a feature F, we define the function $\sqcup_F : 2^{\sigma_F} \to \Sigma_F$ (called the *join* function) as the answer to the single intent inference problem. In other words, the join of a group of behavior is the most specific intent that represents all behavior in that group. In practice (Sec. 3.2.4) we use join on two labels (rather than a group of labels), i.e. $\sqcup_F : \Sigma_F^2 \to \Sigma_F$ where $\sqcup_F(a, b) = \sqcup_F(\sigma_F(a) \cup \sigma_F(b))$.

For most of the feature types in our library, computing join of two labels is straightforward and efficient: $\sqcup_{DAG}(a, b)$ is their least cost common ancestor, $\sqcup_{Flat}(a, b)$ is a if $a = b$ else Any, $\sqcup_{TBV}(a, b) = c$ where $c_i = a_i$ if $a_i = b_i$ else $c_i = x$. $\sqcup_{Range}([a, a'], [b, b']) = [min(a, b), max(a', b')]$, $\sqcup_{Tuple}(a, b) = (\sqcup_{F_1}(a_1, b_1), ..., \sqcup_{F_n}(a_n, b_n))$. For example $\sqcup_{D_{dc}}(U_1, U_2) = User$, $\sqcup_{D_{dc}}(U_1, FW_2) = Any$, $\sqcup_{Range}([1, 2], [2, 4]) = [1, 4]$, $\sqcup_{TBV}(001, 100) = x0x$, and $\sqcup_{F_{dc}}((10.0.1.2, U_1, FW_1, S_1), (10.0.1.3, U_1, FW_1, S_2)) = (10.0.1.2/31, U_1, FW_1, Server)$.

Computing join for $HRE\langle F, d\rangle$ is more complex and requires dynamic programming ($O(d^3|\Sigma_F|^2)$). Here we discuss the high level idea and omit the details. Interested reader can refer to [50] for more details. Given two paths v and w, for any $0 \le i \le |v|$, $0 \le j \le |w|$, $0 \le n \le \min(|v|, |w|)$, $c \in \Sigma$, we define $f(i, j, n, c, m)$ to be the cost of join of a prefix of v with length i and a prefix of w with length j with an HRE of length n where the last label used in the HRE is either c or c^+. Boolean m indicates whether the label at the last position has already been used to match any characters from v/w or not. It is easy to see that f has optimal substructure. After computing f for all values of i, j, n, c, m (in $O(d^3|\Sigma_F|^2)$) we can find the $\sqcup(u, w)$ and its cost from $\min_{1 \le n \le \min(|v|,|w|), c \in \Sigma_F} f(|v|, |w|, n, c)$. For instance, the join of the two paths in Example 2 is $AS_1.R_1.Internal^+.R_5.AS_2$. The idea can be similarly extended for join of two general HREs rather than just paths.

Observation: Because of the way we defined label costs, we can use $\delta(\sqcup(P))$ as a measure of dissimilarity/unrelatedness of the paths in a set P. Higher cost means we have to lose more precision to represent all paths together using a single intent, hence the paths are probably not related to each other, i.e. do not come from the same intent. For example, by comparing $\delta(\sqcup_{D_{dc}}(U_1, U_2)) = \delta(User) = 3$ with $\delta(\sqcup_{D_{dc}}(U_1, FW_2)) = \delta(Any) = 7$ we get that U_1 is more related to U_2 than FW_2, which matches our intuition.

Path selection

Having a solution for single intent inference, the next task is to decide which subset of paths should be fed into the single intent inference problem. Following the observation above, we treat the general intent inference problem roughly as a clustering problem where the goal is to put the more similar paths into the same clusters and then feed these clusters as inputs to the single intent inference problem to infer one intent per cluster. In our clustering, the similarity measure mentioned above can be used to define the distance between paths and clusters of paths.

Specifically our clustering approach is inspired by the Hierarchical Agglomerative Clustering (HAC) technique [51]. In HAC, each object (i.e. path) is considered as a separate cluster initially. The clusters are then iteratively merged with each other until a single cluster remains. At each iteration, the clusters with the minimum distance are selected to be merged with each other. In our approach we terminate the iteration once at most k clusters remain. We define the distance $d(c_i, c_j)$ between clusters c_i and c_j as the amount of increase in the cost of representation by merging the clusters compared with the sum of cost of individual clusters: $d(c_i, c_j) \equiv \delta(\sqcup(c_i \cup c_j)) - (\delta(\sqcup(c_i)) + \delta(\sqcup(c_j)))$. [6]

Optimizations

A naive implementation of the solution described in the previous is not efficient and can not scale. Here we discuss the optimizations that enables Anime to scale to large networks with tens of millions of forwarding paths.

Approximation of join. As a first optimization we approximate $\sqcup(c_i \cup c_j)$ by $\sqcup(\sqcup(c_i), \sqcup(c_j))$[7]. $\sqcup(c_i)$ and $\sqcup(c_j)$ are approximated recursively during clustering. This way, the join function is applied to only two labels at a time rather than to all labels in $c_i \cup c_j$. This reduces the

[6] Although we refer to d as the distance between clusters throughout the chapter, it is not technically a *distance measure*. It's rather a measure of dissimilarity of clusters.

[7] In most feature types and experiments $\sqcup(c_i \cup c_j) = \sqcup(\sqcup(c_i), \sqcup(c_j))$.

complexity of clustering from cubic to almost quadratic (see below).

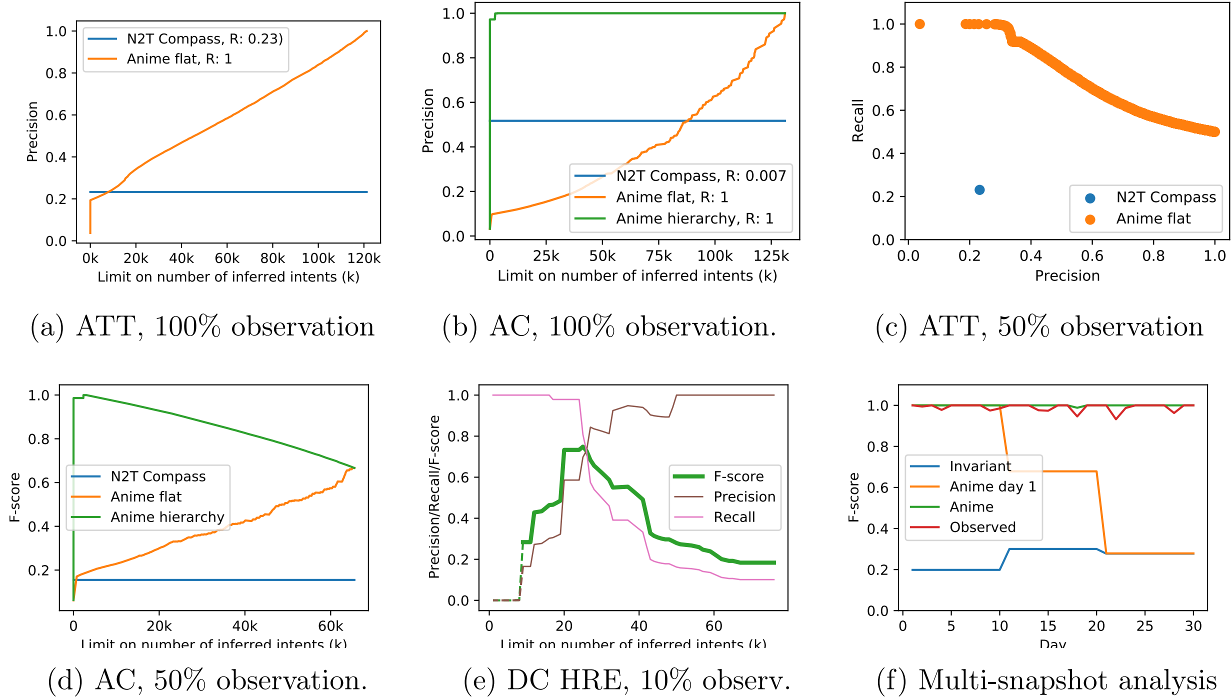
Merging subsets. Each time a new cluster is formed by merging two other clusters, by definition of join, the newly formed cluster may consume more that just those two clusters. Other clusters may also be subsets of the newly formed cluster. We immediately merge all such subsets into the newly formed clusters rather than waiting for the clustering process to merge them one by one.

Parallelization. Before our clustering algorithm selects the first pair of clusters to merge, it need to find the closest pair of clusters among the initial clusters (i.e. the original paths). This involves a nearest neighbor lookup per each path. The set of clusters is fixed during this phase, thus the lookup involves read-only operations. This allows us to easily parallelize the lookup for this phase. Specifically, we use a thread pool of workers, each looking for the closest cluster for a given initial cluster. Parallelization significantly enhances the performance of the initial phase which roughly corresponds to half of the computation during clustering (Sec. 3.3.7).

Indexing clusters. The complexity of a naive implementation of hierarchical clustering in which the set of all clusters is linearly scanned to find the closest cluster (and subsets) per each cluster is $O(J \times N^2 logN)$ where N is the number of input paths and J is the complexity of join (for two labels). An approach with quadratic complexity in the number of input paths cannot scale to large networks. To alleviate this problem, we index the clusters to narrow the search for nearest neighbours (and subsets).

Our index data structure is inspired by R-trees [52], balanced tree data structures often used to index geometric multi-dimensional data. In a d-dimensional R-tree, similar data objects are grouped together and represented by their d-dimensional Minimum Bounding Rectangle (MBR). Each node of an R-tree corresponds to one MBR that bounds its children. Data objects are stored at the leaf nodes. The key idea behind R-trees is that since all children of a node are contained within the MBR of of that node, a query that does not intersect with the MBR of a node, does not intersect with its children either. Insertion is logarithmic in the number of inputs. Subset search (and remove) is logarithmic on average (linear in the worst case). R-Trees for geometric rectangles with euclidean distance have optimized logarithmic algorithms [53] for finding the (k) nearest neighbors of a query.

We adopt R-trees for our application. In our case the data objects are the clusters and the MBRs are defined as the join of the children of a node (i.e. MBRs of children in internal nodes and data objects in leaf nodes). The distance between clusters is the same measure defined in Sec. 3.2.4. Although there is no (proven) theoretical guarantee that the optimized nearest neighbor search algorithms mentioned above would correctly identify the actual nearest neighbor of a cluster in our case, in our experiments we did not observer a visible



(a) ATT, 100% observation (b) AC, 100% observation. (c) ATT, 50% observation

(d) AC, 50% observation. (e) DC HRE, 10% observ. (f) Multi-snapshot analysis

Figure 3.6: Anime experimental results. R stands for recall.

different between the quality of the inferred intents with the optimized algorithm and our reference approach (a linear search through all clusters). Interested readers may refer to [50] for the details of our implementation.

With the index, the average case complexity of our clustering becomes $O(J \times Nlog^2N)$. Our experiments (Sec. 3.3.7) indeed show a significant speedup with the use of index that scales quasi-linearly with the number of input behavior.

3.3 EVALUATION

We implemented two prototypes of our framework, one in 3K lines of C++17 code and one in 1K lines of Python code. The Python implementation includes more features (e.g. implementation of HRE) and is used for comparison with the related work (see below). The C++ implementation is used for larger scale and performance sensitive experiments. As one baseline, we re-implemented Compass, a heuristic algorithm for Net2Text's [43] formulation of the network behavior summarization problem (Chapter 5). Also as a baseline for our multi-snapshot analysis, we implemented invariant detection over multiple snapshots, the main idea behind Invar-net [44] and Config2Spec [45].

In our experiments, we first assess the quality of Anime's inferred intents and compare it

with the related work. Particularly, we study the effect of the algorithm, limit on the number of inferred intents, flat vs. hierarchical labeling, HREs, perfect observation (summarization) vs. imperfect observation (prediction), and single-snapshot vs. multi-snapshot analysis.

Next, we study Anime on a large operational campus network with hundreds of devices and millions of forwarding paths. We evaluate the quality of inferred intents for the summarization use case only as we do not have the ground truth at hand. We also demonstrate the use of Anime in finding anomalous behavior in that network. Finally, we evaluate the performance of Anime on our largest dataset and show the effects of various factors on the performance.

The majority of the experiments ran with the C++ prototype on a machine with Intel Xenon CPU ES-1660 3.30GH with 12 virtual cores and 32GB of DDR3 1333MHz RAM. Unless otherwise noted, the C++ experiments ran with 12 threads in the initial phase of clustering. A few experiments ran with the Python prototype on a Macbook Air, 1.6GHz Intel Core i5, 8GB 1600MHz DDR3 RAM with a single thread.

3.3.1 Comparison with Net2Text

We reuse Net2Text's own evaluation dataset used in [43]. The dataset uses real-world topologies, IPv4 RIB, and AS-to-organization information. It simulates the forwarding state in a simplified ISP network. Each dataset entry contains various information about a path traversing through the ISP from an ingress to an egress. Because Net2Text can not effectively deal with entire forwarding paths or hierarchical values (Chapter 5), we only focus on the ingress device and egress devices and the destination organization of each path to have a fair comparison. Specifically, we use the feature $(organization, ingress, egress)$ as an instance of $\mathit{Tuple}\langle \mathit{Flat}\langle O\rangle, \mathit{Flat}\langle D\rangle, \mathit{Flat}\langle D\rangle\rangle$ where D is the set of network devices and O is the set of organizations. We experimented with various topologies with varying nodes, egresses, and destinations. Figure 3.6a shows a representative result using the AT&T topology with 25 nodes, 5 egresses, and 10k destinations, resulting in more than 121k paths. The x and y axes show the limit on the number of inferred intents (k) and the precision, respectively. Recall rate (same for all k) is reported in the legend. Depending on the limit on the number of inferred intents, Anime achieves better precision than Net2Text. Moreover, Anime's precision increases as we increase the limit. In any case, Anime's recall is 100%. This is due to the fundamental design decision of representing all observed behavior, which is encoded in our problem definition and approach. Net2Text's Compass algorithm has a fixed precision and recall of near 23%. By increasing k, Compass produces new intents that are subsets of previous intents, thus no increase in precision or recall. Compass tries to optimize Net2Text's scoring function which is designed according to Net2Text's goals and assumptions (Chapter 5),

not directly the precision and recall, although the scoring function is related to these two factors in an ad-hoc way.

3.3.2 Effect of Hierarchies

To show the effects of hierarchical values on the quality of inferred intents, we use a dataset resembling resource access control policies. For a network with n endpoints (assuming each is a server in a data-center), we generate random 3-level hierarchy of group names and randomly partition the servers between theses groups as follows. The top of the hierarchy (layer 1) contains one label (*Any*) that represents all servers. Layer 2 consists of of l_2 labels, each being a subset of *Any* and a superset of between min_2 to max_2 labels at layer 3. Similarly layer 3 consists of l_3 labels, each being a subset of a label in layer 2 and containing min_3 to max_3 servers. Labels belonging to the same layer are disjoint from each other.

We then randomly generate m intents each of the form *"server/group A can access server/group B"*. We generate the set of all server pairs represented by the intents as input for our experiment.

We experiment with two types of features for Anime, namely $\mathit{Tuple}\langle \mathit{Flat}\langle D\rangle, \mathit{Flat}\langle D\rangle\rangle$ and $\mathit{Tuple}\langle D_{ac}, D_{ac}\rangle$ where D is the set of servers and D_{ac} is the feature type defined by the DAG representing the hierarchy we described above. We also evaluate Net2Text's Compass.

Figure 3.6b shows the result of a representative experiment with $n = 2000, l_2 = 5, min_2 = 2, max_2 = 10, l_3 = 20, min_3 = 10, max_3 = 200, m = 15$ resulting in more than $131k$ paths.

Anime with flat labeling starts with higher precision than Net2Text (50%) for large limits on the number number of inferred intents (k). By decreasing the limit, Anime flat's precision proportionally decreases, dropping below Net2Text's precision at relatively large values of k (near 90k). On the other hand, hierarchical labeling guides Anime to achieve high precision even for small values of k. The precision remains above 97% for $k \geq 15$ and drops sharply for limits below that (15 happens to be equal to the number of actual intents). In any case, Anime has $>$140x better recall than Net2Text.

3.3.3 Imperfect Observations

We assess Anime's effectiveness in the face of imperfect observations by repeating the previous experiments with random subsets of the original input. We then compute the precision and recall according to the original input. In this setting, the limit on the number of inferred intents (k) is interpreted as parameter one can tune to avoid over/under-fitting.

Figure 3.6c shows the precision recall tradeoff for various values of k in the ATT experiment where only half of the original paths are observed. As expected, an increase in k results in higher precision and lower recall rates. Anime strikes a better balance between precision and recall than Net2Text.

Figure 3.6d shows the results of similar experiments with the access control dataset. The x-axis shows k and the y-axis shows the F-score of the result which is defined as the harmonic average of precision and recall ($= 2 \times Precision \times Recall/(Precision + Recall)$). For high values of k Anime overfits the observations and while it has high precision, its recall rate and thus F-score are low. As k gets smaller, hierarchical labeling achieves better recall without scarifying precision, so it achieves better F-score. Higher recall for flat labeling significantly sacrifices precision, thus its F-score declines. Hierarchical labeling reaches its peak F-score at near $k = 2400$, from there the score remains above of 99.95% until $k = 13$. For values below that, hierarchical labeling starts to underfit and its precision and F-score fall sharply. For the optimal values of k, Anime with flat and hierarchical labeling achieve at least 4 and 6 better F-score than Net2Text, respectively

3.3.4 Experiment with HREs

To showcase the use of HREs we create a synthetic data center topology consisting of c clusters, each containing f firewalls connected to p spine switches which are themselves connected to l leaf switches. Each leaf connects r racks, each containing s servers. The firewalls are connected to g gateway routers shared among the clusters. The gateways are connected to i ISPs providing Internet connectivity[8]. We consider the following actual intents: The servers within each cluster can talk to each other and to the internet. Internet can talk to the servers in a special cluster called DMZ. DMZ cluster servers can talk to servers in any other cluster. The set of possible paths are all shortest paths allowed by the described intents. We take a subset of possible paths where only one random path among all allowed paths between any two points is observed. We feed the observed paths to Anime with feature $HRE\langle D_{ft}, 8\rangle$ where D_{ft} is the hierarchy described above. We run an experiment with $c, f, p, s, g, i = 2; r = 1$ resulting in 75 observed paths (750 possible paths). Below is an example output for $k = 9$ achieving precision and recall of 20% and 100%, respectively.

```
1:Server.Leaf.Spine.Firewall.Gateway.Internet,
2:Internet.Gateway.DMZFirewall.DMZSpine.DMZLeaf.DMZServer,
3:DMZServer.DMZLeaf.DMZSpine.DMZFirewall.Gateway.Cl1Firewall.Cl1Spine.Cl1Leaf.Cl1Server,
4:Cl1Server.Cl1Leaf.Cl1Spine.Cl1Leaf.Cl1Server, 5:Cl1Server.Cl1Leaf.Cl1Server, 6:Cl1Server,
7:DMZServer.DMZLeaf.DMZSpine.DMZLeaf.DMZServer, 8:DMZServer.DMZLeaf.DMZServer, 9:DMZServer
```

[8] The full topology and hierarchy of device labels are available in [54].

The imprecision is due to intents partly representing non-optimal or impossible paths, e.g. 4 includes paths between servers connected to the same leaf that go through a spine. Figure 3.6e shows the precision, recall, and F-score of the results. For $k<9$, the repetition operator appears in some intents (e.g. `Server.Network.Any+` for $k = 3$, `Any.Network.Any+` for $k = 2$, and `Any+` for $k = 1$). Due to the complexity of computing precision in such cases, we estimate it based on Anime's cost (Sec. 3.2.2). F-score is near its peak value (0.75) for $k \in [20, 25]$, and falls as we move away from that range.

3.3.5 Multi-snapshot Analysis

We can use Anime to analyze behavior collected over multiple snapshots of the network. For Anime, time of path observation is simply another feature of a path. To demonstrate this, we reuse our setup for access control experiment and produce a time-series of multiple snapshots that span over 30 days as follows. We first generate our 3-level label hierarchy and initial intents. These intents slightly and randomly change once every 10 days, i.e. on days 11 and 21, r random intents get removed and a random new ones get added. The observed behavior for each day is the set of all behavior represented/allowed by the intents that are in effect on that day plus a small amount of "negative noise": we randomly select a few servers and assume they have failed (each server fails with probability p), so all behavior relating to that server is removed from the set of observed behavior for that day.

We use Anime with feature (day, source device, destination device) as an instance of type $Tuple\langle Range, D_{ac}, D_{ac}\rangle$.

As a baseline we implement the dynamic invariant detection idea: we find the behavior that remain invariant by intersecting the observed behavior among all snapshots. This is the main idea behind Invar-net [44]. Config2Spec [45] can also be viewed as an instance of this idea. Technically Config2Spec finds invariants over link failures, but the high-level idea is still finding properties that do not change (are invariant) over one dimension (feature). As a second baseline we also compare the results with using Anime on only a single snapshot (from day 1).

Figure 3.6f depicts the F-score of the results per each day for a representative experiment with $n = 200, l_2 = 5, min_2 = 2, max_2 = 10, l_3 = 10, min_3 = 5, max_3 = 30, m = 20$, $r = a = 3, p = 0.05$, resulting in near 80k paths spread over 30 days. The results are also compared with the observed behavior on each day. Anime achieves significantly higher F-score than the invariant based method. The precision of the invariant based method is always 1[9] but its recall is low (<20%) as expected since it only considers behavior that exhibit

[9]Unless the observed behavior contains "positive noise", which we do not consider in this dissertation

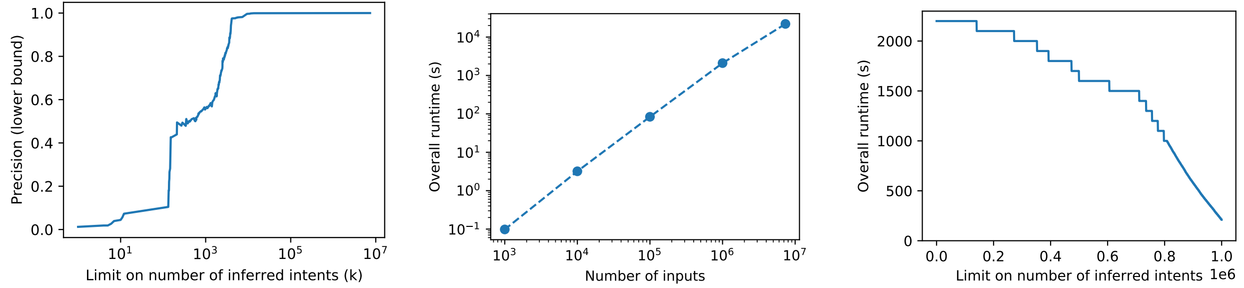
in all snapshots. Moreover, the invariant based method is not resilient to policy changes and "negative noise". On the other hand Anime achieves both high precision and recall (and thus F-score). Comparing the result of Anime with the observed behavior on each day demonstrates again that Anime can tolerate "negative noise". Also contrasting the F-score of Anime given all snapshots vs. a single snapshot shows that Anime can detect and handle policy changes over time on time-series of snapshots, and can distinguish it from noise.

3.3.6 Campus Network Experiments

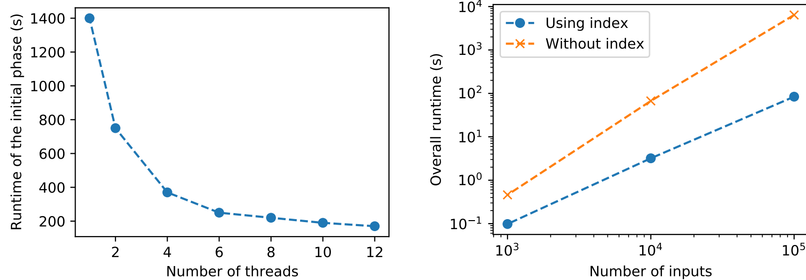
We demonstrate how Anime works in the wild by using our tool on an operational campus network with hundreds of devices. We have access to snapshots of data plane and routers configurations. Due to parsing limitations, we restrict our analysis to a subset of the network consisting of Cisco devices and extract forwarding entries only. The subset consists of 15 core routers connecting 121 building routers with each other and with 12 other routers. Cumulatively, these devices contain nearly 700K forwarding entries. As we do not have access to the physical topology of the network, we use the configuration files to infer the L3 topology by looking at interfaces with overlapping subnets (which is not perfect; §3.3.6). We created a simple symbolic execution tool to extract the end-to-end paths from the data plane snapshots. The symbolic execution starts with a symbolic packet from each building router and traces the packet throughout the network. It stops its exploration and reports a path (along with packet header conditions on that path) when either the packet is delivered at a device, exits the part of network under consideration, is dropped, or a forwarding loop is detected. This results in nearly 12M paths for a single snapshot. As a prepossessing step, we compress the IP addresses as much as possible, resulting in 7.3M paths that we use as input to our tool. We extract the device hierarchy (D_{campus}) from the name of the devices. E.g. `core8-2` is part of the `core8` group which in turn is part of the `core` group. This results in nearly 170 labels in 4 levels. We focus on the paths that go out of the network or are received at a device and encode them by the tuple (destination IP prefix, source device, egress/destination device) of the type $\mathit{Tuple}\langle \mathit{IPPrefix}, D_{campus}, D_{campus}\rangle$.

Summarization

In the first experiment, we summarize the forwarding behavior using Anime. Figure 3.7a shows the precision of results for various limits on the number of inferred intents (recall is always 1). Anime significantly reduces the number of clusters by 3 orders of magnitude with a small (1%) cost in terms of precision. As we shall see in Section 3.3.6, part of this



(a) Precision, 100% observation (b) Scale with #input paths (c) Scale with k



(d) Scale with number of threads (e) Scale with vs. without index

Figure 3.7: Anime results for the campus network experiment and performance experiments

imprecision is actually a result of interesting and anomalous behavior that has led us to interesting observations.

For $k \approx 7M$ to $k \approx 4K$, the precision remains very high (>0.97). From there the precision drops almost linearly with decreasing k, reaching nearly 0.4 for $k \approx 150$. From there, it drops even more steeply to 0.1 for $k \approx 130$, remains almost steady until $k = 12$ and falls again to near 0 for $k = 1$.

The steady $k \approx 7M$ to $k \approx 4K$ phase corresponds to forming new clusters with few or zero false positives. The zero false positive cases mostly correspond to putting multiple paths to the same destination (as result of ECMP routing) together. The cases with a few false (usually one or two) false positives correspond to exceptional behavior (Sec. 3.3.6) for a few IP addresses. The drop in precision starting from around $k \approx 4K$ corresponds to putting source devices together and the sharp drop starting from around $k = 150$ corresponds to putting destination/egress devices together. From $k = 12$ to 1, clustering starts to merge completely irrelevant behavior.

Anomaly analysis

We can use Anime to detect anomalies in the forwarding behavior as follows. During the clustering process, whenever the precision of a cluster is below 1, it means that part of the

inferred behavior is not observed. This can be either due to grouping irrelevant behavior together, or it can be a true indicator of a behavior that should exist but does not (assuming perfect observation). So flagging the predicted but unobserved behavior, especially for large values of k with high precision, may discover incorrect or interesting behavior.

To demonstrate this, we first summarize all behavior for packets starting from a specific building. We then compute the precision of each cluster and for clusters that have below 100% precision, identify the set of behavior predicted by that cluster that is not observed in the input. We only look at imprecise clusters created as a result of merging a pair of clusters with 100% precision. We manually inspect these behaviors in the order of creation of their respective clusters. During our experiments, we identified 4 general groups of such behavior, all leading to interesting observations.

- The first group of such behavior are due to the existence of forwarding loops for some IP addresses. Upon further investigation it turned out that the apparent loops are a bi-product of imperfect topology inference – i.e. two interfaces in two routers having overlapping interface subnets but not being actually connected physically.

- The second are the ones in which all but a few IPs end up going out of a certain device and those few go out of another device. Investigating these behaviors revealed a bug in the implementation of the symbolic execution code related to the semantics of forwarding rules.

- The third are rules such as the example below in which the first IP address in a block of IP addresses is dropped, the second IP address is received by the device, and the rest are forwarded to another device.

```
Prefix          Next hop        Interface
x.y.z.128/25    Attached        VlanA
x.y.z.128/32    Drop            Null0
x.y.z.129/32    Receive         sup-eth
```

  Interestingly, there is no line in the configuration of these devices that directly corresponds to "x.y.z.128/32". We followed up with the administrators and they were not completely sure of the reason for this behavior. Our guess is that such behavior is related to the convention of not using the first (and the last) IP addresses in a subnet.

- The last and the most interesting group are a few public /32 IP addresses that have static Null0 (dropped) routes. We asked the network administrators about these rules.

They mentioned the rules were originally added due to a request from the security team. They also said the rules were very old (12 years) and should have been removed if they were unnecessary; This prompted the administrator to follow up with the security team.

The fact that we made these interesting observation merely through analysis of similar behavior, without any prior knowledge about the network or even knowing what we were after, demonstrates the effectiveness of our general approach.

3.3.7 Performance

Anime's performance is significantly influenced the by number of input paths and the complexity of its features. It is also slightly influenced by the limit on the number of inferred intents (k). Here we evaluate these effects. We also show how our optimizations contribute to the scalability of our tool.

Effect of input size. To show scalability with the number of inputs, we randomly sample the input in the campus network experiment (Sec. 3.3.6) with various sample sizes and use the results as input to Anime. Figure 3.7b shows the log-log plot of the overall runtime vs. number of inputs for a complete run of our algorithm with $k = 1$. Thanks to our optimizations (Sec. 3.2.4), the runtime has an log-linear relation with the number of inputs. Anime is able to summarize the full dataset with >7.3M entries in near 6h, i.e. 3ms per input on average.

Effect of limit on the number of inferred intents (k). Figure 3.7c plots the runtime with respect to the limit on the number of inferred intents (k) on a random sample of 1M paths from the campus network. The runtime increases with the decrease of k, with a gentle slope of 2ms per unit decrease of k. There is a constant 210s factor that corresponds to indexing (40s) and the initial phase (Sec. 3.2.4) of clustering (170s). Note that the initial phase runs in parallel with 12 threads, but the rest run sequentially.

Effect of feature complexity. The hierarchical feature causes a 2x slowdown in the access control experiment. Also the data center HRE experiment is 1.2K times slower than an experiment with the same input size for ATT as HRE join is relatively expensive.[10]

Effect of optimizations. We focus on the effect of non-trivial optimizations: indexing and parallelization. Figure 3.7d shows the runtime of the initial phase with various numbers of threads for a random sample of 1M paths from the campus network. The runtime decreases significantly with the number of threads. E.g., the initial phase with a single thread takes 1400s, amounting to more than 40% of the overall runtime (for $k = 1$). On the other hand, the initial phase with 12 threads takes only 120s, nearly 7% of the overall runtime.

[10]The Python implementation was used in this experiment.

Indexing significantly speeds up our algorithm. It effectively turns our quadratic algorithm to a linear one (Sec. 3.2.4). To show the effect, we turn off indexing and resort to linear scan over all clusters in the campus network data-set with various input sizes. In Figure 3.7e, the orange plot shows the overall runtime (for $k = 1$) without using an index and the blue plot corresponds to use of an index. Anime without index scales poorly (i.e. quadratically) with the input size, taking roughly 105 minutes for 100K inputs while Anime with index would take approximately 1 minute.

3.4 DISCUSSION

Expressiveness. Anime's expressive power depends on the expressiveness of the feature type that is used to encode the behavior. Abstractly, Anime can express and infer intents that can be represented as a disjunction of clauses that could be encoded using the labels of that feature type. For example, this formalism can express/infer important classes of functional intents including reachability and waypointing under various (temporal, topological, header, etc.) conditions. On the other hand, Anime is not designed to directly infer negative behavior such as in isolation intents. This can be alleviated by encoding negative behavior as positive behavior, e.g. packet drop as reachability to a special node, or using labels representing complemented sets (e.g non-firewall or $!\{\texttt{TCP}, \texttt{UDP}\}$).

As mentioned in § 3.2.3, packet modification along a forwarding path can be expressed and inferred with appropriate feature design. Say, by using $\langle\langle \text{src device}, \text{src packet header}\rangle$, $\langle \text{dst device}, \text{dst packet header}\rangle\rangle$ as an instance of an appropriate feature type, given the input behavior, Anime could infer intents such as $\langle$src:$\langle$*Internet*, (130.5.0.0/16, `HTTP`, URL:`"/a/*"`)$\rangle$, dst:$\langle$*Server group X*, (10.0.3.0/24, `RPC`, procedure:`"A.Service.*"`)$\rangle\rangle$. By using or extending our library of feature types, one could design more complex types capable of capturing even more sophisticated modifications such as encapsulation.

While Anime's approach is well suited and optimized for inferring network-wide intents, it is not best suited for inferring arbitrary program logic. Inferring such is typically the target of more classic inductive program synthesis research (e.g., [55]). The approaches proposed in that line of work focus on synthesizing programs from a small set of input-output behavior examples and can not scale to networks with millions of forwarding paths.

Feature engineering and parameter tuning. We do not necessarily expect the end-user (network operator) to be in charge of these tasks. A front-end layer between Anime and the end-user – designed for specific application, network type, collector, etc. – can abstract the low-level details, though the user may be given direct/indirect (see below) control for better results. The layer can also employ automatic techniques for feature selection/parameter

tuning – e.g. in summarization, it can suggest promising values for k using the elbow method; for prediction, k can be tuned via cross-validation.

3.5 CONCLUSION

Anime enables a novel approach towards bridging the semantic gap between high-level intents and low-level network behavior by inferring the former from the later. We provide a formal setup that fits the hierarchical nature of networks and enables application of data mining techniques to network behavior. We use the setup to define the intent inference problem and provide a heuristic solution based on hierarchical clustering with a suite of optimizations that allows our approach to scale. Our experiments with various synthetic and real-world datasets demonstrate the scalability of our approach and its effectiveness in inferring high-quality intents even in the face of imperfect observations and policy changes, producing compact summaries, and flagging interesting anomalous behavior.

Chapter 4: Modeling: Formal Semantics of P4 and its Applications [3]

Traditionally, to handle the network scale, the networking hardware has been hard coded with well-established network protocols needed to run and manage the network. However, doing so has the downside of not being able to cope with the speed of innovation that is necessary to satisfy the diverse and growing set of user demands, because the process of modifying networking equipment tends to be slow and expensive. This has ignited a line of research whose goal is to make networks more programmable.

One of the most recent developments in this line of research, P4 [22], is a high level declarative programming language for programming packet processors. P4 allows the developers to specify how a packet processor should process its incoming packets. A P4 compiler then translates the P4 program into an instruction set understandable by the target hardware. The examples of targets include software switches, high performance ASICs, FPGAs, and programmable NICs.

Since its introduction in 2014, P4 has attracted significant interest because the flexibility that it provides enables rapid development of a diverse set of applications that can potentially work at line rate, such as In-Band Network Telemetry [56] and switch based implementation of Paxos [57]. However, this flexibility, combined with the complexity of networks and networking hardware, increases the chance of introducing subtle bugs that are very hard to discover manually, yet can have catastrophic effects, from service disruptions to security vulnerabilities.

Even without P4, answering the simplest questions about the correctness of a network (e.g., what kind of packets can reach node B from node A) has become manually prohibitive when the scale and complexity of networks is taken into account. Subsequently, a large body of research has recently focused on automating the process of network verification [5, 6, 7, 9]. However, most of these works assume a simple fixed structure for the packet processors and, as a result, may miss many details. P4 makes manual verification even harder, if not impossible. Consequently, there is a big need for automated tools to analyze P4 programs or networks of nodes programmed using P4.

We adhere to [58] that analysis tools for any programming language must be based on the formal semantics of that language rather than on its informal specification. Informal semantics are subject to interpretation by different tool developers and usually there is no guarantee that these interpretations are consistent with the specification or with each other. As shown in [58], state-of-the-art program analysis tools based on informal language specifications "prove" incorrect properties or fail to prove correct properties of programs due to

their misinterpretation of the semantics of the target programming language. Moreover, the informal language specification itself might have problems, such as ambiguities, inconsistencies, or even parts of the language not defined at all. This is particularly relevant for new languages, like P4, whose design has not matured yet.

It is therefore important to develop a formal semantics for P4. Furthermore, to build confidence in the adequacy of a formal semantics, we believe it should be: (1) *executable*, so it can be rigorously tested against potentially hundreds of programs; (2) *compact and human readable*, so it can be easily inspected and ultimately trusted by everyone. Finally it must be (3) *modular*, so new language features can be formalized without the need to change the previously formalized features.

To this end, we have developed P4K, an executable formal semantics of P4 based on the official P4 language specification [59]. P4K faithfully formalizes all of the language features mentioned in the P4 specification, with a few exceptions corresponding to features whose meaning was ambiguous or incorrect or under specified and we did not find any satisfactory way to correct it. We have reported some of these issues to the P4 language designers [60, 61, 62, 63, 64, 65, 66, 67, 68] and are working on a modified version of the specification [69] addressing the issues. We validated P4K by executing 40 test cases provided by one of the official compiler front-ends of P4 [70], a manually crafted test suite of 30 tests, and by formally analyzing several programs.

We chose the K framework [23] for our P4 formalization effort. It has several advantages that make it a suitable choice. First, a language defined in K enjoys all three properties mentioned above. Second, once a programming language semantics is given, K automatically provides various tools for the language, including an interpreter and a symbolic model checker, at no additional effort. Finally, K has already been successfully used to formalize the semantics of major programming languages such and C [71], Java [72], JavaScript [73], etc.

The focus of this work is the P4K formalization of P4, but we also show how P4K and the tools provided by K can be used beyond just a reference model for the language. We discuss several applications useful for P4 programmers, language designers, and compiler developers, such as: detection of unportable code, state space exploration of P4 programs and of networks, bug finding using symbolic execution, data plane verification, deductive verification, and translation validation. Specifically, we make the following contributions:

- P4K: the most complete formal semantics of P4, based on the official specification of $P4_{14}$ version 1.0.4.
- A collection of P4 formal analysis tools for the networking domain, derived directly from our semantics.

The rest of the chapter is organized as follows. § 4.1 overviews P4 and K, as well as the challenges in defining a semantics for P4. § 4.2 describes P4K, our K semantics of P4, and discusses some of problems that we identified in the language specification. § 4.3 evaluates our semantics. In § 4.4 some of the applications of the semantics are discussed.

4.1 BACKGROUND AND CHALLENGES

Here we give background on P4 and K. We also discuss some of the challenges that we faced in formalizing P4.

4.1.1 Software Defined Networks

Control plane is the part of the network responsible for making packet forwarding decisions by running computations (e.g. routing algorithms) based on the network state. *Data plane* is the collection of forwarding devices (or packet processors) that actually carry the network packets and execute the forwarding decisions. Traditionally, each device had its own vendor-provided control plane hard-coded on the device. The need for rapid innovation has sparked interest in Software Defined Networks (SDNs). SDN is a modern architecture in which the control plane is physically separated from the data plane. In this architecture, one controller can *program* a set of forwarding devices through open, vendor-agnostic interfaces such as OpenFlow [74].

In OpenFlow, each device processes the packets according to the contents of one or more *flow tables*. Each table will contain a set of *flow entries*. Abstractly, each entry is a *(match, action)* tuple. *Match* provides values for specific fields in the packet header, and *action* denotes the action to be performed if the packet header matches the respective values in match. Possible actions include dropping, modifying, or forwarding the packet. The controller programs the data plane through installation and modification of flow entries.

OpenFlow assumes a fixed structure for the forwarding devices. It explicitly specifies the set of protocol headers on which it operates, the structure of the flow tables, the set of possible actions, etc. Modification to any of these features requires an update to the OpenFlow specification. Over the course of 4 years since the initial version of OpenFlow, the number of supported header fields in its specification has been more than tripled [22].

4.1.2 P4

The limitations of OpenFlow and the need for expressiveness has lead to the introduction of P4, a high level declarative programming language for expressing the behavior of packet

```
header_type h_t {
  fields { f1 : 8; f2 : 8; }
}
header h_t h1;
parser start {
  extract(h1);
  return ingress;
}
action a(n) {
  modify_field(h1.f2, n);
  modify_field(standard_metadata.egress_spec, 1);
}
action b() {
  modify_field(standard_metadata.egress_spec, 2);
}
table t {
  reads { h1.f1 : exact;}
  actions { a; b; }
}
control ingress {
  apply(t);
}
```

Figure 4.1: A very simple P4 program

processors. In P4, one can program a custom parser for their own protocol header, define flow tables with customized structure, define the control flow between the tables, and define custom actions. Hence, P4 allows the developers to specify how a packet processor should process its incoming packets (note, however, that P4 does *not* provide a mechanism to populate the flow table entries; this is done by the controller). A P4 compiler then translates the P4 program into the instruction set of the hardware of the packet processor on which the program is installed (the *target*).

We briefly describe the basic notions of the language here. § 4.2 discusses the language construct in more details. A P4 program specifies at least the following components [59]. *Header types:* Each specifies the format (the set and size of fields) of a custom header within a packet. *Instances:* Each is an instance of a header type. *Parser:* A state machine describing how the input packet is parsed into header instances. *Tables:* Each specifies a set of fields that the table entries can match on and a set of possible actions that can be taken. *Actions:* Each (compound action) is composed of a set of primitive actions which can modify packets and state. *Control flow:* Describes the custom conditional chaining of tables within the packet processor's pipeline.

For example, in Figure 4.1, `h_t` is a header type consisting of two 8 bit fields `f1` and `f2`. Header `h1` is an instantiation of `h_t`. The parser consists of a single state `start` which

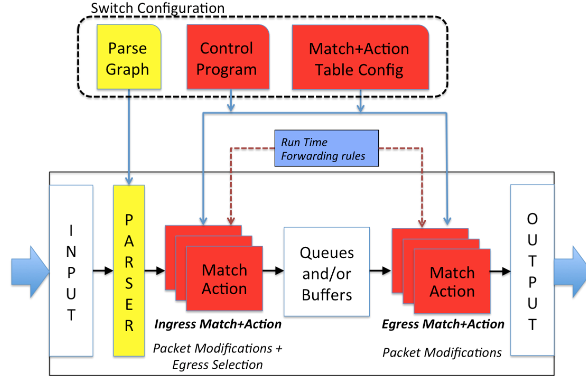


Figure 4.2: $P4_{14}$ Abstract Forwarding Model [59]

extracts `h1` from the input packet. There are two compound actions `a` and `b` in the program. The actions use the `modify_field` primitive action. The entries in table `t` match on field `f1` in `h1` and if applied, may call actions `a` or `b`. An entry calling `a` must provide a value for `n`. The `ingress` pipeline in this program only consists of applying table `t`.

P4 programs operate according to the abstract forwarding model illustrated in Figure 4.2. For each packet, the parser produces a *parsed representation* comprised of header instances and sends it to the *ingress* match+action pipeline. Ingress, among other things, may set the *egress specification* which determines the output port(s). Ingress then submits the packet to the queuing mechanism the specification of which is out of the scope of P4. The packet may also go through the optional *egress* pipeline which may make further modifications to the packet. Finally (if not dropped) the parsed representation will be *deparsed* (i.e serialized) into the packet which will be sent out. P4 also supports re-circulation (looping packets inside the device) and cloning of packets.

The P4 Language Consortium (`http://p4.org`) provides the official specification of the language, as well as various other tools including compiler front ends, software interpreters, runtime and testing infrastructure, etc. There are two versions of P4 in current use, $P4_{14}$ and $P4_{16}$. $P4_{16}$ has been released by the consortium in May 2017 [75] and it is much simpler and cleaner than $P4_{14}$, at the cost of deliberately breaking backwards compatibility with $P4_{14}$. There are, however, important $P4_{14}$ program which have not been translated to $P4_{16}$, and, indeed, the $P4_{14}$-to-$P4_{16}$ translator provided by the consortium is not semantics-preserving [76]. Ideally, we would like to prove the translator correct, but for that we need formal semantics of both $P4_{14}$ and $P4_{16}$. In this work we only discuss our formal semantics of $P4_{14}$, leaving that of $P4_{16}$ and the correctness of the translator as future work. Throughout the section, we refer to $P4_{14}$ simply as P4.

4.1.3 Challenges in Formalizing P4

P4 has several characteristics that make it a challenging target for formalization. Here we discuss some of this challenges and the way we dealt with them.

Unstable language: P4 is a relatively young language and it takes time for the community to reach consensus on its design. When we started, the only publicly available version of P4 was $P4_{14}$ v. 1.1.0. That version soon was deprecated and replaced with v. 1.0.3 which we initially used to develop our semantics. In the middle of our formalization effort, $P4_{16}$ v. 1.0.0 as well as a minor revision of $P4_{14}$ (v. 1.0.4) were released. Thanks to K's support for modular definitions and reuse, we were able to rapidly adapt to changes and finalize our semantics according to $P4_{14}$ v. 1.0.4. Through continuous discussions with the P4 designers, we hope to help them reach more stable versions sooner.

Imprecise specification: Since P4 is a newly developed language its specification is not free of problems. There are many inconsistencies and corner cases which are not discussed (§ 4.2). One of the important contributions of this work is the identification of these problems through rigorous formalization. We have reported some of the problems to the community. We are also working on a modified version of the specification [69] addressing the issues we found.

No comprehensive test suite: Similar formalization efforts for other languages (e.g. [71, 73, 77]) rely heavily on official test suites. Unfortunately, there is no official test suite for P4. To alleviate this problem, in addition to testing our semantics against a test suite we obtained from a P4 compiler, which only covers about half of our semantic rules (§ 4.3), we hand-crafted a test suite that gives a complete coverage of our semantic rules.

Unconventional input: The input to a P4 program is different from that of conventional programming languages. P4 has two sources of input. One is the stream of incoming packets that the device running the P4 program needs to process. The other is the table entries and configurations that are installed by the controller at runtime. The mechanism by which the controller interacts with the target at runtime is device-specific and is therefore out of the scope of the language specification. Still, to be able to execute and analyze P4 programs, for the target-specific language features we tried to provide the most unrestricted executable semantics; for example, if the order of some operations was unspecified then we chose a non-deterministic semantics, so we can still explore the entire state-space of behaviors using the K tools.

We also grouped most of the target specific semantic rules in a separate semantic module. This way, the semantics is parametric on the target specific details. One can provide a new target specific module to change the target specific behavior, without the need to touch the

rest of the semantics. We have already used this feature when we were testing our semantics against the p4c test suit as it contained target specific features and assumptions (§ 4.3).

4.1.4 The K Framework

K [23] is a programming language semantics engineering framework based on term rewriting. Its underlying philosophy is that tools for a language can and should be automatically derived from the formal semantics of that language. Indeed, K provides an actively growing set of language-independent tools, i.e., tools which are not specific to any language but apply to any language which has a K formal semantics. These include a parser, an interpreter, a symbolic model checker, a sound and (relatively) complete deductive program verifier, and, more recently, a cross language program equivalence checker and a semantic-based compiler. Some of the tools are useful during the formalization process itself, the most important of which is the interpreter. Using the interpreter, the semantics can be tested against potentially many programs to gain confidence in its correctness.

To define a programming language in K, one needs to define its syntax and its semantics. Syntax is defined using BNF grammars annotated with semantic attributes. Semantics is given using rewrite rules (also called semantic rules) over configurations. A configuration is a set of potentially nested cells that hold the program and its context. Each cell contains a piece of semantic information of the input program such as the its state, environment, storage, etc. Semantic rules are transitions between configurations: if parts of the configuration match its left hand side, rewrite those parts as specified by the right hand side. Figure 4.3 shows a rule taken from P4K (modified for presentation) concerning reading the value of field `F` from instance `I` as an example.

$$\left\langle \frac{I.F}{V} \cdots \right\rangle_{\mathsf{k}} \; \langle \langle I \rangle_{\mathsf{name}} \; \langle true \rangle_{\mathsf{valid}} \; \langle \cdots \; F \mapsto V \; \cdots \rangle_{\mathsf{fieldVals}} \cdots \rangle_{\mathsf{instance}}$$

Figure 4.3: A rewrite rule capturing the semantics of reading a field in P4

The contents inside each matching pair of angle braces constitutes a cell, with the cell name as subscript. The `k` cell contains the list of computations to be executed. The fragment of computation at the front of the list (the left most) is executed first. There are multiple `instance` cells each corresponding to a header instance. `name` contains the name of the instance and `valid` keeps its validity state. `fieldVals` is a map from each field name to the value stored in the field in the given instance. The ellipsis are part of the syntax of K and denote contents irrelevant to the rule. The horizontal line denotes a rewrite. If

the configuration matches the pattern, the part of the configuration above the line will be replaced by the content below the line. The rest of the configuration remain intact. A rule may contain multiple rewrites at different positions of the configuration. In that case, all rewrites will be applied in one step.

This example illustrates two properties of K that makes it suitable for giving semantics specially to evolving programming languages like P4. First, note that the actual configuration contains many more cells and each cell may contain multiple elements, but the rule only mentions the cells that are relevant. The *configuration abstraction* feature of K automatically infers what the rest of cells should be. Second, note that rewrites are local. There is no need to rewrite the whole configuration. These two features make K rules succinct and human readable. More importantly, they enable modular development of the semantics: if the language specification adds or modifies a language feature the rules irrelevant to that feature do not need to be modified.

4.2 P4K

P4K is the most complete executable formal semantics of $P4_{14}$. It is based on the latest (at the time) official language specification (v. 1.0.4 [59]) and on discussions with the language designers. Our work is open source and is available online [3]. The formalization process took 6 months to complete by a PhD student with some familiarity with the K framework. Most of the time was spent learning K and understanding the details of the P4 specification, including its problems. P4K contains more than 100 cells in the configuration, 400 semantic rules, 200 syntax productions, and 2000 lines.

4.2.1 Syntax

The language specification provides a BNF grammar, whose conversion to K was straightforward. We mostly copy-pasted the grammar and made a few minor modifications to make it compatible with K.

During this process, in addition to minor problems, we identified [60] an ambiguity in the syntax between the minus sign in a constant value (for specifying negative constant values) and the unary negation operator. This ambiguity has important semantic effects. In P4, all the field values have a bit width associated with them. According to the specification "For positive [constant] values the inferred width is the smallest number of bits required to contain the value". Also "For negative [constant] values the inferred width is one more than the smallest number of bits required to contain the positive value" [59]. So for example -5

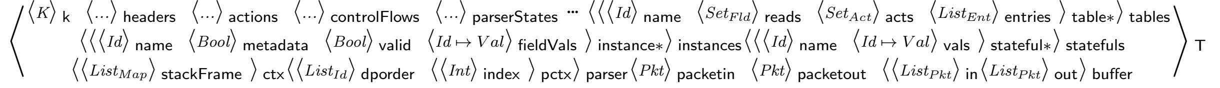


Figure 4.4: Part of the P4K configuration. The ellipsis symbols indicate omitted cells.

interpreted as a negative constant would yield a 4 bit value while if interpreted as negation of a positive constant would yield a 3 bit result. Used in an expression with other operators, this difference may affect whether the expression overflows or not, which subsequently may affect the final result.

4.2.2 Configuration

The configuration contains more than 100 cells. Figure 4.4 shows part of it, featuring more important cells. All of the language constructs including headers, instances, parser states, actions, tables, control flows, etc have respective cells in the configuration containing their static information and/or runtime state. For example the `tables` cell will contain a set of `table` cells ("*" denotes multiple cells with the same name). Each of the `table` cells contains a table's static information such as its `name`, the fields to match (`reads`), and possible actions (`acts`). It also contains runtime information such as the `entries` installed in the table.

Some cells contain the execution context. For example during the execution of an action, the `stackFrame` cell holds a stack of maps from each formal parameter of the executing action to the respective argument values passed to the action.

The cells `in` and `out` contain the input and output packet stream from/to all ports respectively. `packetin` contains the current packet being processed and `packout` contains the packet being serialized.

Cells are populated or modified by processing the input P4 program before execution, during the initialization, or during the execution as discussed next.

4.2.3 Semantics

After parsing, the P4 program populates the `k` cell and is executed with the semantics rules.

Execution Phases The rules describe the P4 program execution, in three phases.

Preprocessing: In this phase, P4K iterates over all the declarations in the input P4 program (in the `k` cell), creating and populating the corresponding cells and preparing the

configuration for execution. In some cases auxiliary information is pre-computed for the execution phase. An important such computation is the inference of the order of packet headers for deparsing. Details will be discussed in Section 4.2.3.

Initialization: There is an optional initialization phase after preprocessing. It is used primarily to prepopulate the tables and packet buffers before the execution in certain analysis such as symbolic execution. The tables and packet buffers can also be populated at runtime in normal execution.

Runtime: The actual execution of a P4 program happens during this phase. It implements the abstract forwarding model. Packets are taken from the input packet stream and processed using the entries installed in the match+action tables by going through the ingress and potentially the egress pipelines. The output packets are appended to the output packet stream. This phase never terminates.

Language Constructs and Semantics We briefly describe the language constructs and primarily focus on interesting findings and relevant semantics.

Header types: Each header type is a named declaration that includes an ordered list of fields and their attributes (e.g. field width and signedness). P4 also allows declaration of variable length headers. During our formalization, we found corner cases (e.g. [64]) in which the semantics of such headers are not completely clear.

Instances: Instances may be referenced in various runtime stages including parsing, table matching, and action execution. Some instances, called *header instances* (although the naming is not consistent throughout the specification [65]), keep the parsed representation of the respective packet headers (i.e., the packet header is extracted into the header instance). Other instances, called *metadata*, keep arbitrary per packet state throughout the pipeline. For example `h1` in Figure 4.1 is a header instance and `meta` in Figure 4.10 is a metadata. In our semantics, both types of instances are kept as `instance` cells, distinguished by their `metadata` cells (Figure 4.4).

It is also possible to declare fixed size, one dimensional array instances (called *header stacks*), as sequences of adjacent headers (e.g. to support MPLS [78]). We keep array elements as separate header instances, with special names that include their index. Otherwise, the elements are treated same as other instances.

Header instances are invalid (uninitialized) until validated in parsing or by specific primitive actions in match+action processing. According to the specification, reading an invalid header results in an undefined value, whose behavior is target dependent. We model this using a special value `@undef`. Use of `@undef` in an expression or action call causes the execution to get stuck by default. We use this feature to detect unportable code (Section 4.4.1).

Hash generators: The ability to calculate a hash value for a stream of bytes has various uses in networking. P4 provides the ability to declare hash generators (called *field list calculations*). The developer provides a list of values (declared using a *field list* struct) and selects a hash generation algorithm. The hash generator computes the hash of the bitstream generated from the list. In the example in Figure 4.5 `ipv4_checksum` is a hash generator for the `ipv4_checksum_list` field list (declaration omitted) with the IPv4 checksum algorithm (`csum16`).

```
field_list_calculation ipv4_checksum {
    input { ipv4_checksum_list; }
    algorithm : csum16;
    output_width : 16;
}
```

Figure 4.5: A *field list calculation* example

The language specification identifies a set of well known hash generation algorithms (e.g., IPv4 checksum and CRC). In our semantics, we treat hash generation as a black box; K allows us to "hook" library function calls that implement the desired functionality. It is possible to also directly specify the algorithms using K rules inside the semantics, but we did not find any compelling reason to do so.

We found a problem [67] with the specification during the formalization of field lists. Each element of a field list can refer to a field in an instance, an instance itself (when all the fields in that instance are used), another field list (when all the fields identified by that list are used), a constant value, or the keyword `payload`. According to the specification "payload indicates that the contents of the packet following the header of the previously mentioned field is included in the field list" [59]. However, "previously mentioned field" is ambiguous. For instance, in Figure 4.6 it is not clear if payload refers to `f1` or `f2`.

```
field_list fl1 { h.f1; }
field_list fl2 { h.f2; fl1; payload; }
```

Figure 4.6: Ambiguity in the semantics of `payload`

Thus we do not provide semantics for payload. $P4_{16}$ has replaced field lists with a C-like *struct* construct, disallowing the payload keyword.

Checksums: A field in a header instance can be declared to be a *calculated field*, indicating that it carries a checksum. The developer provides a hash generator for verification of the checksum at the end of parsing, and/or an update of the checksum during deparsing. For

example, in Figure 4.7, the field `hdrChecksum` in the header instance `ipv4` is declared to be a calculated field which uses the hash generator `ipv4_checksum` for its verification and update.

```
calculated_field ipv4.hdrChecksum  {
    verify ipv4_checksum;
    update ipv4_checksum;
}
```

Figure 4.7: A *calculated field* example

The P4 specification leaves undefined the order in which the calculated fields must be updated or verified. For verification, the order can matter depending on the target. For update, the order can matter in cases where the field list calculation of a calculated field includes another calculated field. After discussing [68] with the language designers, to obtain the most general behavior, we decided to choose a non-deterministic order for update and verify. K provides a search tool which one can use to explore all possible non-deterministic outcomes to check whether they differ. (Section 4.4.2).

Parser: The user can define a parser to deserialize the input packet into header instances (the *parsed representation*). The parser is defined as a state machine. In each state, it is possible to *extract* header instances (i.e., copy the data from the packet at the current offset into respective field values for the given instances) and to modify metadata. Then, it is possible to conditionally transition to another state, to end the parsing, or to throw an (explicit) exception. For example, in state `parse_ethernet` in Figure 4.9, after extracting the `ethernet` header, based on the value of the `etherType` field, the parser may transition to the parser state `parse_ipv4` or end the parsing and start the ingress pipeline.

Exception Handlers: P4 allows us to declare exception handlers for implicit or explicit parser exceptions. In case an exception occurs, parsing is terminated and the relevant handler is invoked. Each handler can either modify metadata and continue to ingress or immediately drop the packet. There is a default handler that drops the packet. For example in Figure 4.1 if a packet is too short for the extraction of `h1`, an implicit exception is thrown and the default handler drops the packet.

Deparsing: This is the opposite of parsing. At egress, the (potentially modified) valid header instances are serialized into a stream of bytes to be sent. An important question in deparsing is the order in which the header instances should be serialized. The parsing order is not enough to find the deparse order, since header instances might also be added (validated) or removed (invalidated) in the match+action pipeline. According to the specification "[A]ny format which should be generated on egress should be represented by the parser used on ingress" [59] and the order of deparsing should be inferred from the parse graph.

If the parse graph is acyclic, a topological order can be used as the deparsing order. However, in general the graph might be cyclic as there may be recursion in parsing. While in simple cases an order can still be inferred (e.g., cases where recursion is only used for the extraction of header stacks), there are cases in which a meaningful order can not be inferred. This is a well known problem [79]. In our semantics, we support simple cases of cyclic parse graphs. All of the practical examples we have seen so far can be handled by our semantics.

$P4_{16}$ has switched to an approach in which the deparse order is explicitly defined by the programmer.

Stateful Elements: P4 supports *stateful* language constructs that can hold state for longer than one packet, as opposed to per packet state in instances. *Counters* count packets or bytes, *meters* measure data rates, and *registers* are general purpose stateful elements. The declaration of each of these elements creates an array of memory units. The units may be *direct*ly bound to the table entries. In that case, the (counter and meter) units will automatically be updated when the corresponding entry is matched in match+action. The units may alternatively be *static*; then they should explicitly be accessed or updated via special primitive actions. For example, in Figure 4.10, `reg` is a static register with a single 8 bit memory cell.

In our definition we unified all these elements as instances of the `stateful` cell (Figure 4.4) which can be accessed like registers. Other operations are defined as functions which read and manipulate the registers. Each `stateful` cell in the configuration has a map from an index to a value. The index is either a table entry id (for direct) or an array index (for static). The mechanism of updating meters is target specific (not part of the language specification). Subsequently we do not perform any action in case a meter is updated. If needed, one can add a mechanism in our target specific module.

The specification does not specify [62] the initial value of the stateful elements. It is sensible to assume that the initial value of counters is 0, and similarly for meters. For registers, by default we initialize the registers to `@undef`. Moreover, the specification is inconsistent [63] about whether direct meters are allowed to be explicitly updated by table actions. To be consistent with counters and registers, we assume they are allowed.

Finally, if multiple counters/meters are directly bound to the same table, the specification does not state [68] the order in which the elements must be updated when an entry in that table is matched. The order can affect the outcome in multi-threaded packet processors (Section 4.2.4) as there may be data races over stateful elements. Again, we choose a non-deterministic order for updating the counters/meters, so we can systematically explore it using K's search.

Actions: *Compound* actions are user defined imperative functions that can take arguments

and if called, perform a sequence of calls to other compound actions or built-in *primitive actions*. Primitive actions provide various functionality including arithmetic, addition/removal/modification of instances and header stacks, access/modification of stateful elements, cloning, re-circulation, dropping the packet, etc. Actions are executed as a result of table matches.

We formalized all the primitive actions (see Section 4.2.5 for limitations on clone primitive actions). The specification does not specify the behavior of some corner cases, such as shift with negative shift amount. We intentionally do not provide semantics for such cases to detect unportable code.

The previous version of the specification [80] stated that all primitive actions resulting from a table match execute in parallel, making the semantics of the action in Figure 4.8 unclear.

```
action a() {
  modify_field(h.f, 1);
  modify_field(h.f, 2);
}
```

Figure 4.8: Unclear semantics of parallel action execution

`modify_field(f,v)` is a primitive action that updates field `f` with `v`. The latest revision (1.0.4) switched to sequential semantics, so we do not have to deal with this case anymore.

Tables: Tables will be populated at runtime by the controller. Each entry provides values for the fields that are specified in the declaration, an action that should be executed if the entry is selected, and arguments to be passed to the action.

The interaction mechanism between the controller and P4 target is out of the scope of the specification. Hence, the answer to questions such as what happens if a table is modified by the controller while it is being applied on a packet is target dependent. We currently assume that modification and application of the same table are mutually exclusive.

P4 provides various matching modes per each field. For example, *exact* matches exact numbers and *ternary* matches ternary bit vectors. It is also possible to associate priorities with table entries. In case more than one table entry is applicable, the rule with the highest priority will be selected. *Longest Prefix Match (LPM)* is a special kind of ternary match useful for IP prefixes. The specification specifies how the relative priority of an entry with LPM match can be inferred bases on the corresponding match value of the entry. However, it does not specify how the priority should be decided in cases where there are more than one field with LPM match type [61]. We assume all entries have explicit (unique) priorities regardless of their match types. We keep the entries sorted in their descending order of

priority. To apply a table on a packet, we iterate over the entries in order and select the first matching entry.

Control Flow: User defines the order and the conditions under which various tables are applied to a packet using *control functions*. The body of a control function is a control block consisting of a sequence of control statements. A statement might apply a table, call a control function, or conditionally select a control block. Ingress is a special control function that is automatically called after (successful) parsing. Egress is another (optional) special control function. If defined, it will automatically be called when the queuing mechanism takes the packet to be sent out.

Other constructs: We omit the discussion of *value sets*, *action profiles*, and *action selectors* as well as many details of the discussed constructs. Interested readers can refer to the semantics [3] for more details.

4.2.4 Concurrency Model

Real world high performance packet processors have multiple threads of execution. The specification is silent about the concurrency model. As a result, what constitutes a thread depends on the target hardware. In our semantics, we support a multithreading model in which each thread individually does all of what a single threaded program does by addition of a few more cells and rules[1] (similar to Section 4.2.6). The input/out packet streams, the tables, and the stateful elements constitute the shared memory between the threads.

4.2.5 Limitations

P4 provides four primitive actions for cloning a packet under process from ingress/egress to ingress/egress. The actions that clone a packet into the egress put the clones in the queue between ingress and egress pipelines. Since we currently do not model the ingress and egress pipelines as separate threads, we only support a single packet in the queue between the two. Therefore, we do not directly support clone into egress. Instead, we treat such clones as new incoming packets with auxiliary flags to skip the ingress pipeline.

4.2.6 Network Semantics

It is useful to be able to simulate or analyze a network of P4 programs rather than just a single program (Sections 4.4.4 and 4.4.2). In order to do so, we need the semantics of

[1] In all of our experiments we used only a single thread for each P4 program. Throughout the section we assume executions are single threaded.

the network. Thanks to the modularity of K, we easily modeled the semantics of a P4 network without changing the P4 language semantics. We only needed to add a few more cells and preprocessing rules. We added a root `nodes` cell containing multiple `node` cells each containing the configuration of a P4 program plus a `nodeId` cell. We also added a `topology` cell which holds the connection between the nodes.

To model the network links, we added a single rule that takes a packet from the end of the output stream of one node and puts it at the beginning of the input stream of the node it is connected to. If needed, one can also model packet loss in the links by a single additional rule. Note that here we have multiple threads of concurrent execution, whose interleaving is non-deterministic. The thread interleaving space can be explored using the K search mode (Section 4.4.2).

4.3 EVALUATION

K provides us with an interpreter derived automatically from the semantics, enabling us to test our semantics. Official conformance test suits are an ideal target for testing executable semantics. Unfortunately, P4 does not have such a test suite. A new official P4 compiler front end (p4c [70]) has a limited set of tests for $P4_{14}$, which we used in our evaluation.

Generally, it is non-trivial to port tests across different implementations of P4, as its IO is not specified (Section 4.1.3). Fortunately, the p4c tests were easy to adopt. Each test, along with the P4 program under test, contains an STF file. The file describes table entries, input packets, and the expected output packets. We systematically converted the STF files into our test format.

The suite contains tests with minor issues including use of deprecated syntax, unspecified constructs, or unspecified primitive actions. We fixed the issues by slightly modifying to the corresponding P4 programs or test files, and implementing the primitive actions in a target specific module for the tests. Moreover, the tests assume undefined[2] egress specification leads to packet drop. The specification does not specify the behavior in this case, so it is target dependent. In our semantics, by default the execution gets stuck in such cases. In our target dependent module for the tests, we added a rule to drop the packet in such cases.

The tests also helped us identify a few problems in P4K. For example, we found that we had misunderstood the semantics of a primitive action (`pop`). Note that `push` and `pop` have rather an unusual semantics in P4 [66].

[2] According to the specification egress specification is undefined unless set explicitly. We model this using the `@undef` value.

After fixing the problems with the tests and our semantics, P4K passed 39 out of the 40 test. The failing test[3] has multiple inferable deparsing orders. The order chosen in our execution happens to be different from the order the test expects. We verified that both orders are possible.

Inspired by [73], we measured the percentage of the semantics rules exercised by the tests (the *semantic coverage* of the tests). The tests cover under 54% of the semantics and miss many of the semantic features. We have also manually developed 30 tests during our formalization process. Together, these 70 tests cover almost all the semantic rules.

Each test took 19.5s (± 3.2s) on average with the maximum of 125s.[4] We note that approx. 10s out of this time is the startup time of K and is not related to execution. We also note that K has multiple backends. We use an open source backend [58] which is relatively very poor in terms of performance. We expect the runtime to improve by orders of magnitude on performant commercial backends (e.g. [81]).

4.4 APPLICATIONS

Besides defining a formal semantics for P4 and thus helping make the P4 specification more precise, a secondary objective of our effort was to make use of the various tools that K provides. We demonstrate how the tools can be useful for the P4 developers and network administrators, as well as for the P4 language designers and compiler developers.

4.4.1 Detecting Unportable Code

As seen above, in some cases the P4 specification does not provide the expected behavior of the program. P4 programs exhibiting such unspecified behavior may not be portable among different targets and compilers. It is not wise to solely rely on the expertise of P4 developers in the low level details of the specification to check if their code is portable. It is desirable to have tools that automate this check. For simple cases, such behavior may be detectable by syntactic checks. In general, unspecified behavior may depend on the input.

By default, we do not provide semantics for cases which are not covered by the language specification. If the execution of a program reaches a point with unspecified behavior, the execution gets stuck. Avoiding over-specification therefore allows us to check for unspecified behavior in P4 programs. This is done simply by running the program and checking whether

[3] Namely `parser_dc_full.stf`.

[4] All experiments are run on a machine with Intel Xenon CPU ES-1660 3.30GHz and 32GB DDR3 1333MHz RAM.

it reaches a state in which it gets stuck or not. The check can be performed using either concrete or symbolic inputs. We show a symbolic example in Section 4.4.3.

To tune the semantics for a specific target, one can provide custom semantics for cases with unspecified behavior in the target specific module.

4.4.2 State Space Exploration

K provides a *search* execution mode which allows us to explore all possible execution traces when non-determinism is present. In K, non-determinism occurs when more that one rewrite rule is applicable, or the same rule is applicable at multiple positions in the configuration. In normal execution mode, only one of the applicable rules is (non-deterministically) selected. In the search mode, all the applicable rules are explored. Moreover, the user can explicitly control the points in which non-determinism is explored. This allows one to focus on exploration of one or more specific sources of non-determinism and ignore the rest.

There are two sources of non-determinism in P4K. The first is due to our approach to model the most general behavior. Examples are order of deparsing, order of update and/or verification of calculated fields, and order of update of direct stateful elements. The second is due to the existence of multiple threads of execution. These include the threads of execution inside a single P4 program, as well as the execution of multiple nodes in a P4 network. Both sources can be explored using the search mode. We have already shown in § 4.3 how exploration of the order of deparsing can be useful. The benefits of exhaustive analysis of thread interleavings in concurrency analysis are well known.

4.4.3 Symbolic Execution

K allows the configuration to be symbolic – i.e., to contain mathematical variables and logical constraints over them. During execution with a symbolic configuration, K accumulates and checks (using Z3 [35]) all the logical constraints over the execution path – i.e the conditions under which the rules are applicable to respective states. Under the hood, there is no difference between symbolic and concrete execution. Symbolic execution powers some of the other K tools such as the program verifier (Section 4.4.5) and the equivalence checker (Section 4.4.6). It can also be useful on its own, say, to search for bugs in P4 programs and data planes.

Search for Bugs To illustrate one application, we choose a community provided sample P4 program which defines a very basic L3 router [82]. Using symbolic execution, we find input

packets for which the program fails to specify the egress specification, leading to unspecified behavior.

To do so, we prepopulate the tables with entries from the unit test provided along with the program. We then simply start the program with a single symbolic packet (P) from a symbolic port in the input packet stream (the `in` cell). Our goal is to find an input packet that leads the program to a state in which neither packet is dropped, nor its egress specification is set. We run the program in the (symbolic) search mode. The search returns multiple inputs which can lead to undefined egress specification. Here we only discuss one of the more interesting ones: the search result suggests that if "P has ethernet as its first header and ethernet.etherType != 0x0800", then the program will end up with an undefined egress.

```
...
parser start {
  return parse_ethernet;
}
parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    0x0800 : parse_ipv4;
    default: ingress;
  }
}
control ingress {
  if (valid(ipv4)) {
     ...
  }
}
```

Figure 4.9: Part of a basic L3 router [82]

Figure 4.9 shows the relevant snippet of the program. A simple manual inspection confirms the finding. The parser extracts the `ethernet` header and checks `etherType`. If it is equal to 0x0800 (i.e the IPv4 ether type), the parser then proceeds to extracting the `ipv4` header (not shown). Otherwise, instead of, say, dropping the packet, the program starts the ingress pipeline. At the beginning of the ingress, the program checks the validity of the `ipv4` header. If valid, the pipeline applies a sequence of tables that may set the egress specification (not shown). Otherwise the program does not apply any tables and the egress specification remains undefined. Thus, under the given constraints, packet is not dropped, `ipv4` is invalid, and the egress specification is undefined.

4.4.4 Data Plane Verification

There is a growing interest towards *data plane verification* tools such as [5, 6, 7, 8, 9]. These tools analyze the table entries in a snapshot of the data plane and look for violation of

properties of interest. The verification of these properties usually requires answer to queries of the following form: *What kind of packets from node A will reach node B*? While using various smart ideas to achieve better performance, all these tool are based on the same basic idea: symbolic reasoning over the space of packet headers.

Using our semantics, we can answer such queries by inserting a symbolic packet at, say, node A and using symbolic execution to find the constraints on the packets that end up at node B. The tools mentioned above use simplified hardcoded/adhoc models of packet processors in their analysis and miss the internal details of such devices. They need to be re-engineered to change their model of packet processors. There is no such need in our case. Moreover, as will be shown in the next section, these tools can verify a very restricted class of properties. We eliminate these limitations.

4.4.5 Program Verification

K features a language independent program verification infrastructure based on Reachability Logic [58]. It can be instantiated with the semantics of a programming language such as P4 to automatically provide a sound and relatively complete program verifier for that language. In this system, properties to be verified are given using a set of reachability assertions, where each reachability assertion is written as a rewrite rule. A reachability assertion asserts that starting from any configuration matching the left hand side of the assertion, by execution using the input semantics, one will either eventually reach a configuration that matches the right hand side of the assertion or never terminate.

The standard pre/post conditions and loop invariants used in Hoare style program verification can be encoded as reachability assertions. Intuitively, a Hoare triple $\{P\}C\{Q\}$ becomes "$C \wedge P$ *rewrites to* $. \wedge Q$" where "." is the empty program [77].

The Load Balancer Program To showcase the use of the program verifier, we provide a simple P4 program and verify a simple property about it.

The program in Figure 4.10 is meant to balance its incoming packets (from any port) between two output ports. This is done using a register whose value alternates between 0 and 1 across incoming packets. The program features a single register, a metadata instance, and two tables. The parser starts ingress without extracting anything. We install a single entry in `read_reg_table` to call action `read_reg`. The action copies the register value (at index 0) into `meta.reg_val` [5]. We install two rules in `balance_table`. One rule matches if `meta.reg_val = 1` and calls `balance(1,0)`. The other rule matches if `meta.reg_val = 0`

[5]This is done because register values can not directly be matched in tables.

```
header_type meta_t {
  fields { reg_val : 8; }
}
metadata meta_t meta;
parser start {
  return ingress;
}
register reg {
  width: 8;
  instance_count: 1;
}
action read_reg() {
  register_read(meta.reg_val,reg,0);
}
table read_reg_table {
  reads{ meta.valid : exact; }
  actions{ read_reg; }
}
action balance(port,val) {
  modify_field(standard_metadata.egress_spec,port);
  register_write(reg, 0, val);
}
table balance_table {
  reads{ meta.reg_val : exact; }
  actions{ balance; }
}
control ingress{
  apply(read_reg_table);
  apply(balance_table);
}
```

Figure 4.10: A simple load balancer

and calls `balance(0,1)`. `balance(p,v)` modifies the register (at index 0) with value `v` and effectively sends the packet to port `p`. Our goal is to prove that this program (along with its table entries) correctly balances the load. Specifically, we want to prove the following:

Property: For any input stream of packets, after processing all the packets, no packet is dropped and no new packet is added; all the packets in the output are either sent to port 0 or port 1; and the absolute difference between the number of packets sent to ports 0 and 1 is less than or equal to 1. Albeit simple, none of the data plane verification tools mentioned above are capable of proving it. They lack either support for stateful data plane elements or support for reasoning over an unbounded (i.e symbolic) stream of packets.

In K, the property is captured by the reachability assertion in Figure 4.11. For presentation purposes we have omitted the less relevant, mostly static parts of the specification which hold the program and the table entries. The full specification can be found in [3].

$$\left\langle \frac{@execute}{@end} \right\rangle_{\mathsf{k}} \left\langle \langle reg \rangle_{\mathsf{name}} \left\langle 0 \mapsto \frac{0}{_} \right\rangle_{\mathsf{vals}} \right\rangle_{\mathsf{stateful}} \left\langle \frac{I}{.} \right\rangle_{\mathsf{in}} \left\langle \frac{.}{?O} \right\rangle_{\mathsf{out}}$$

$$\begin{array}{ll} & |onPort(?O,0) - onPort(?O,1))| \leq 1 \\ \text{ensures} & \wedge onPort(?O,0) + onPort(?O,1)) = size(?O) \\ & \wedge size(?O) = size(I) \end{array}$$

Figure 4.11: A reachability assertion capturing the desired property

In the specification, `@excute` is the program state right before the execution starts. `@end` is state after all the input packets are processed[6]. I is a (universal) symbolic variable representing the input packet stream and $?O$ is an existential symbolic variable representing the output stream. Symbol "." in both `in` and `out` cells represents an empty packet stream. The rewrite in the `vals` cell says that the value of the register `reg` (at index 0) is 0 at start[7], and its value at the end is not relevant to the assertion. The keyword `ensures` adds logical constraints on the right hand side of the assertion (i.e the post condition). Function `onPort(s,p)` returns the the number of packets in stream s belonging to port p. Function `size(s)` returns the length of stream s.

Our semantics of P4 contains a main loop over the stream of input packets. Since in our property the input is a symbolic list with an unbounded length, similar to Hoare logic, we need a loop invariant. To prove our property, we provide the loop invariant as the reachability assertion in Figure 4.12.

$$\left\langle \frac{@nextPacket}{@end} \right\rangle_{\mathsf{k}} \left\langle \langle reg \rangle_{\mathsf{name}} \left\langle 0 \mapsto \frac{R}{_} \right\rangle_{\mathsf{vals}} \right\rangle_{\mathsf{stateful}} \left\langle \frac{I}{.} \right\rangle_{\mathsf{in}} \left\langle \frac{O1}{?O2} \right\rangle_{\mathsf{out}}$$

$$\begin{array}{ll} & (R = 1 \wedge onPort(O1,0) = onPort(O1,1)) + 1 \vee \\ \text{requires} & R = 0 \wedge onPort(O1,0) = onPort(O1,1))) \\ & \wedge onPort(O1,0) + onPort(O1,1) = size(O1) \\ & \| onPort(?O2,0) - onPort(?O2,1)) \| \leq 1 \\ \text{ensures} & \wedge onPort(?O2,0) + onPort(?O2,1) = size(?O2) \\ & \wedge size(I) + size(O1) = size(?O2) \end{array}$$

Figure 4.12: The loop invariant

Here, `@nextPacket` is the head of the main loop over the input packet stream. Keyword `requires` puts logical constraints on the left hand side of the assertion (i.e the precondition). The assertion reads as: starting from the head of the main loop, given the constraints in

[6]This state is only added due to technical reasons for verification purposes, as actual P4 programs never terminate. We add a rule causing the program to jump to this state once the input packet stream becomes empty.

[7]We made the assumption that registers are initialized to 0.

`requires` are satisfied, if the program terminates, it will reach an `@end` state that satisfies the constraints in `ensures`.

We gave the two assertions to the K's program verifier instantiated with our P4 semantics. The verifier successfully proved the loop invariant and the first reachability assertion (i.e., the desired property). The verification took about 80s.

In this example, we used concrete table entries as the entries are part of the functionality that we aimed to verify. In general, depending on the property, the tables – as well as anything else – can be symbolic. Weassumedn example in thinitializedtion.

4.4.6 Translation Validation

P4 programs eventually need to be compiled into the instruction set (i.e the language) of the target hardware for execution. With any compilation, there is the question of whether or not the semantics of the input program is preserved by the compiler. Currently the compilers usually lack formal semantic preservation guarantees since providing such guarantees requires a significant effort. The issue is even more pronounced when sophisticated compiler optimizations are involved. A promising alternative approach is to verify each instance of compilation instead of the whole compiler. This approach, known as *tranlation validation* [83], aims to verify the semantic equivalence of a program and its compiled counterpart, potentially using hints from the compiler.

Recently, K has introduced a prototype tool (named KEQ [84]) for cross language program equivalence checking using a generalized notion of bisimulation. The notion enables us to mask irrelevant intermediate states and consider only the relevant states in comparing two program executions. KEQ takes the K semantics of the two programming languages, two input programs written in the respective languages, and a set of synchronization points as input, and checks whether or not the two programs are equivalent.

Each synchronization point is a pair of symbolic states (called *cut*s) in the two input programs. The meaning of synchronization is defined by the user as a logical constraint over the given pair of symbolic states. It usually consists of checking the equality of certain relevant values. Each cut in the pair is essentially a pattern over the configurations of the semantics of the respective languages (similar to the right or left hand side of rewrite rules). The user labels one or more synchronization points as *trusted*. These points are assumed to already be bisimilar. Usually one (and the only one) such point is the end of the two programs and the constraint is the equality of the respective output values.

For the rest of the synchronization points, the equivalence checker checks whether the given points are bisimilar. It basically means that starting from the two cuts in a synchronization

point, using the semantics of the respective languages, all reachable synchronization points are respectively bisimilar. Normally one such point is the start of the two programs. The constraint is the equality of the respective input values. Additional synchronization points may be needed as well, such as the beginning of unbounded loops. We refer the interested readers to [84] for more details on KEQ.

P4 → IMP+ Translation Validation We illustrate KEQ through a small example. We check the equivalence of a simple P4 program with a program written in another language. For this purpose, we developed a very simple imperative language called IMP+. The language syntactically resembles C, although semantically it is much simpler. We also developed the semantics of IMP+ in K. We provide a set of API functions for the language to send and receive packets, read tables, etc. The name of these functions are prefixed with the "#" symbol. For simplicity, we directly provide semantics to such functions in our semantics. We chose the simple P4 program in Figure 4.1 for translation. Figure 4.13 shows our manual translation of it into IMP+.[8]

The goal is to prove the equivalence of the two programs. Our notion of equivalence is defined as follows: for any input stream of packets, and for any table entries in table t, at the end of processing all the input packets, the two programs generate the same output stream of packets. To do so, we manually provide a few synchronization points. We have annotated the IMP+ program with the points. Next we informally describe the points and their constraints. The full specification can be found in [3].

p_0 is the start of the two programs. The condition associated with this points is the equality of the respective input streams, table entries, and table default actions.

p_1 is the main loop over the input packets. Its condition is same as p_0's condition plus the equality of the respective current output packet streams.

p_2 is the loop over table entries. The condition is p_1's condition plus the equality of the field values in the parsed representation of the P4 program and the corresponding variables in the IMP+ program, plus the equality of the index of iteration of the tables entries[9], and the equality of the current packet payloads.

p_3 is the end of execution[10]. The condition is the equality of the respective output packet streams. p_3 is *trusted.*

Figure 4.14 illustrates the abstract transition relation between the points. Each arrow represents multiple rewrite steps in each program, ignoring the irrelevant (possibly non-

[8] We assume packets with undefined egress specification will be dropped.

[9] Note that in our semantics of both P4 and IMP+, the table entries are sorted in the descending order of their priority.

[10] For technical reasons we assume both programs terminate once the input packet stream becomes empty.

```
int h1_f1; int h1_f2; bool h1_valid;
int sm_egress_spec;
bool parse(){
  return start();
}
bool start(){
  if (! #has_next(8))
    return false;
  h1_f1 = #extract_next(8, false);
  if (! #has_next(8))
    return false;
  h1_f2 = #extract_next(8, false);
  h1_valid = true;
  return true;
}
void a(int n){
  h1_f2 = n;
  sm_egress_spec = 1;
}
void b(){ sm_egress_spec = 2; }
void apply_t(){
  //p2
  while (#get_next_entry()) {
    if (#entry_matches(h1_f1)){
      #call_entry_action();
      return;
    }
  }
  if (#has_default_action())
    #call_default_action();
}
bool process_packet(){
  #reset();
  sm_egress_spec = -1;
  h1_valid = false;
  if (! parse())
    return false;
  if (sm_egress_spec == -1)
    return false;
  return true;
}
void deparse(){
  #emit(h1_f1);
  #emit(h1_f2);
  #add_payload();
}
void main(){ //p0
  //p1
  while (#get_next_packet()){
    if (!process_packet()){
      #drop();
    }else{
      deparse();
      #output_packet();
    }
  }
}
//p3 [trusted]
```

Figure 4.13: Manual translation of the P4 program in Figure 4.1 into IMP+

equivalent) intermediate states of the programs. Note how this abstraction enables us to establish the equivalence even though the programs are written in two quite different programming languages.

Given the synchronization points, KEQ was able to prove the equivalence. Although the program is very simple, we believe it captures the essence of many of the programs that are used in practice. In addition, note that we provided the synchronization points by hand. In practice, the compiler can automatically provide this information as it has enough knowledge during the translation.

4.5 CONCLUSION

We have presented P4K, the first complete semantics of $P4_{14}$. Through our formalization process, we have identified many problems with the language specification. We provided a suite of analysis tools derived directly from our semantics. We have discussed and demonstrated

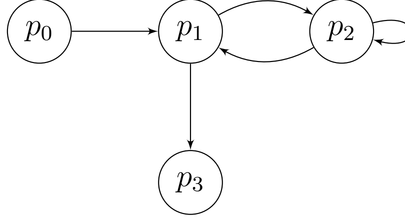


Figure 4.14: Abstract transition relation between p_0, p_1, p_2, and p_3.

the applications of some of the tools for P4 developers and designers.

With the introduction of $P4_{16}$, $P4_{14}$ may sooner or later be deprecated, especially because $P4_{16}$ addresses many of $P4_{14}$'s issues through backwards-incompatible changes. Nevertheless, we think that formalizing $P4_{14}$ was a worthwhile effort. There are still important applications written in $P4_{14}$ (e.g. [85]) that do not have a $P4_{16}$ equivalent. The language consortium provides a translator from $P4_{14}$ to $P4_{16}$. However, without a clear semantics of $P4_{14}$, the translation itself might be problematic. We are aware of at least one instance [76] in which the translator's $P4_{16}$ output is not equivalent to its $P4_{14}$ input. We also plan to formalize $P4_{16}$ in near future. We believe transition to $P4_{16}$ will be straight forward. $P4_{16}$ has actually a smaller core language compared to $P4_{14}$.

Chapter 5: Related Work

5.1 NETWORK VERIFICATION

Bugs happen frequently in networks and lead to performance problems, service disruptions, security vulnerabilities, etc. Scale and complexity of networks make answering even the simplest functional correctness queries prohibitively hard to answer manually. This has ignited research into automating the process of network verification.

The literature in this field is vast and includes BGP configuration checking (e.g., [12, 86, 87, 88, 89, 90, 91, 92, 93]), ACL misconfiguration detection (e.g., [94, 95]), firewall checking (e.g., [96, 97, 98, 99]), SDN verification (e.g., [13, 100, 101, 102]), testing (e.g., [103, 104, 105, 106, 107]), debugging (e.g., [108, 109]), differential analysis (e.g., [110]), concurrency analysis (e.g [111, 112]), automatic repair (e.g., [113, 114, 115]), synthesis (e.g. [4, 116]), programming languages (e.g. [3, 41, 117, 118]), safe network updates (e.g., [119, 120, 121, 122]), data plane checking (e.g., [7, 15, 123, 124]), real-time checkers [5, 6, 9, 125], and more general network analyses (e.g., [10, 46, 92, 126, 127, 128]) together with suitable levels of abstractions (e.g., [129, 130]). The contributions of this dissertation have applications in virtually all of the directions above.

Our analysis framework (#PEC) belongs to a class of methods that partition the space of packet headers before starting the analysis. Other notable works in this class are ddNF [14], APV [8], Delta-net [5] and VeriFlow [6], which are discussed in Chapter 2.

5.2 FACILITATING SPECIFICATION

Net2Text seeks to summarize network traffic "as much as possible" [43]. The formalized version of this problem can be seen as a special case of Anime's where the only allowed feature type is of the form $Tuple\langle Flat\langle S_1\rangle, ..., Flat\langle S_n\rangle\rangle$. Net2Text assigns a *score* to each summary (= set of inferred intents in Anime) that essentially awards representing more paths/traffic and using labels other than *Any*. The score is proportional to precision and recall in an ad-hoc way. The paper provides an approximate algorithm (Compass) that, for a limit on the length of description, produces a summary with maximum score. Unlike Anime, Net2Text cannot handle hierarchies, i.e. it can not choose any value in between a concrete label and *Any*. For instance, between 10.0.1.2 and 10.0.1.3, Compass either chooses one, hence miss the other, or chooses none (effectively $Any = 0.0.0.0/0$) which is too coarse, whereas Anime can represent both together as 10.0.1.2/31. As demonstrated in § 3.3, this

limitation significantly affects the quality of inferred intents. Net2Text cannot handle entire paths either. These limitations prevent Net2Text from effectively representing intents such as the ones in our motivating examples.

Invar-net [44] (an extension of Delta-net [5]), detects networking reachability invariants by observing the reachability relation among network devices for each header equivalence class [5] and intersecting this relation over snapshots of the network obtained over time (or after link failures). The same basic idea has also recently been used in Config2Spec [45]. These works are inspired by software specification mining literature, particularly dynamic invariant inference tools like Daikon [131]. In some sense, Anime is a generalization of this idea to multiple dimensions (e.g. devices, header fields, time) rather than just time or number of link failures, and with the addition of hierarchical values. For instance [44, 45] cannot infer any higher-level information from a single snapshot like in Example 1, while Anime can. Also as shown by our experiments, unlike Anime, invariant based methods can not tolerate negative noise or policy change.

5.3 MODELING AND P4 ANALYSIS

5.3.1 Semantics of Programming Languages in K

The K framework has been used to provide complete executable semantics for several programming languages. Here we briefly overview the more relevant work.

KCC [71] formalizes the semantics of C11, passing 99.2% of GCC torture test suit, more than what the GCC and Clang could pass. Later work [132] develops the "negative" semantics of C11 and is able to identify programs with undefined behavior. In P4K, we identify unspecified behavior by lack of semantics.

K-Java [72] formalizes Java 1.4 and follows a test-driven methodology to manually provide a suite of 800+ tests.

KJS [73] provides semantics for JavaScript passing all 2700+ tests in a conformance test suite [133]. The authors introduce the notion of semantic coverage for test suites which has inspired our work.

In these works, the language design predates the formalization effort by several years. Consequently, although more complex, these languages are quite stable. P4 is still at the early stages of the language design process and is relatively unstable. This made the formalization effort challenging.

KEVM [77] formalized the Etherium Virtual Machine [134], successfully passing a suite of 40K official tests. Like P4K, KEVM targets a new language and reveals problems in its

specification.

These works (except K-Java) rely heavily on existing tests to provide semantics. In our case, such a comprehensive test suite did not exist. The only test suite that we found (at the time) covered less than 54% of our semantics.

5.3.2 Semantics and Analysis of P4

P4NOD [17] provides a big step operational semantics of a subset of the P4 language (on paper). The authors use the semantics to provide a translator from P4 to Datalog. The result is used in P4 data plane verification using a Datalog engine optimized for this purpose [124]. The authors also use the tool to catch a class of bugs, called well-formedness bugs, that are unique to P4 networks. Finally, the authors show an example of P4 to P4 equivalence check.

Vera [18] translates P4 into SEFL, a language designed for network verification, and uses symbolic execution (with SymNet [135]) to uncover bugs including parsing errors, invalid memory accesses, and loops and tunneling errors. Vera focuses on the efficiency of verification using specialized data structures. The authors of Vera provide big step operational semantics of a small subset of the P4 language as well as the SEFL language (on paper) and use it to prove the the correctness of their translation.

P4v [19] translates P4 into the Guarded Command Language (GCL), and enables deductive verification by generating verification conditions from the resulting GCL program and a set of user-provided pre-conditions and post-conditions capturing the property to be verified. The formulas are then checked using Z3. In p4v, the assumptions about the control plane are encoded as symbolic constraints provided via a domain specific language. The authors of p4v do not provide explicit semantics for P4. The semantics is defined by the translation. In order to gain confidence in the translation, the resulting GCL code is symbolically executed to generate a suite of tests. The tests are then verified against the official P4 compiler and a software simulator.

P4pktgen [20] takes an intermediate representation of a P4 program generated by the P4 compiler as input and uses symbolic execution to generate test cases for the given program. The tool only supports a subset of the P4 language features.

P4-assert [21] provides a language for annotating P4 programs with assertions capturing the desired properties. It then translates the intermediate representation of the given program into a C-like program and uses symbolic execution (with KLEE [136]) to uncover assertion failures.

In P4K, we focused on the modeling of the language itself and its problems. We provided a modular small step operational semantics for **all** language features of P4. Using K tools,

among other things, we too are able to perform all the analysis that the other tools perform. In addition, we have also shown an example of translation validation between P4 and an arbitrary programming language formalized in K. Unlike the rest of these works, the analysis is directly based on the formal semantics.

Chapter 6: Conclusion

In this dissertation, we argued that existing network (data plane) verification tools have significant shortcomings in the main ingredient of formal verification (modeling, property specification, and verification data structures and algorithms) that limit their practicality. We then provided foundations for each of these ingredients, addressing the shortcoming in previous works. Specifically, we provided a symbolic packet analysis framework that resolves the tension between efficiency and expressiveness. We then provided a framework for automatic inference of high-level intents from low-level network behavior which can facilitate the process of property specification for network operators. Finally, we provided a rigorous foundation for modeling of network data plane elements by defining a complete operational semantics of the P4 language and deriving a set of formal tools directly based on the semantics. We believe our contributions provide a solid foundation for practical network verification.

6.1 FUTURE WORK

"Now this is not the end. It is not even the beginning of the end. but it is, perhaps, the end of the beginning" [137]. Here we briefly overview some of the interesting future directions for our work.

6.1.1 Verification Framework

Data Structures for On-Demand Analysis. We argued that PEC-based symbolic analysis significantly improves the efficiency of network verification and provided a practical PEC-based framework. The downside of PEC-based analysis is the initial cost of computing the PECs, which as shown in § 2.3.5 can be considerable. This makes PEC-based tools not the best fit for use cases where the user wants to quickly check a light query (e.g. what happens to packets with a particular destination IP address entering a particular switch) as soon as she runs the tool on a snapshot of the network. Answering such a query requires looking at a very small subset of the packet header space and devices. In such scenarios, the benefits gained from using PECs will be shadowed by the initial cost of computing the PECs. This is in contrast with heavier use cases (e.g. would any packet experience a forwarding loop anywhere in the network), where the PEC-based tools shine. Through our interaction with actual network operators, we realized that both use cases are common. We believe a more traditional symbolic analysis approach in which the symbolic constraints are accumulated

and resolved online (during the analysis) by a BDD-like data structure would be a better fit for such on-demand scenarios. However, as shown in Chapter 2, BDDs incur significant per bit overhead. One promising future direction would be to use the domain properties to make BDDs more efficient for network verification. For example, instead of using 64 graph nodes to encode an IPv6 address, one can compress all these nodes (if there is no branching) into a single node that contains a 64-bit value, requiring $C + 1$ instead of $C \times 64$ machine words of overhead where C (≥ 3 in a typical implementation) is the memory overhead of a single node in a BDD.

Quantitative and Probabilistic Analysis. In this work, we discussed intents relating to graph reachability in the data plane, such as loop freedom, isolation, path consistency, and way-pointing. Our analysis excluded the so-called *quantitative* aspects of networks such as load, latency, and bandwidth. Network operators are equally interested in checking properties related to these aspects: E.g. "Is traffic belonging to certain QoS class guaranteed certain bandwidth/latency" or "Would any of the links get overloaded under my traffic assumptions". Moreover, probabilities are inherent in some aspects of the network, such as link or device failures. Future network verification should support reasoning about the quantitative and probabilistic aspects. Early works in this direction [138, 139, 140, 141] are still in their infancy and limited in what they can offer (§ 6.1.4). Improving the practicality of this space is an interesting future direction.

6.1.2 Intent Inference

Interactive intent inference. An interesting future direction is to make the inference process more interactive: the system proposes some intents, and the user selects intents that make sense or marks the ones that do not seem correct. The system then infers new intents that respect the user feedback for example by tuning label costs, etc.

We believe that the hierarchy of inferred intents created during the hierarchical clustering process in Anime might be useful for such an interaction. The operator can inspect the tree at the highest levels first, and if something does not seem right, she can inspect the clusters below that node. She might either find a problem with the network behavior or realize that the lower-level clusters are incorrectly grouped. In the latter case, she can mark the incorrectly grouped clusters. The clustering algorithm can use this feedback for better clustering.

Positive anomaly detection. We demonstrated the use of Anime to detect negative anomalies: missing behavior that are expected to be present. We showed analyzing such anomalies leads to interesting observations. A similarly interesting future work is to detect

positive anomalies: present behavior that are not expected to be present. We believe our framework allows for the application of classic anomaly detection techniques for detecting such behavior, which we leave as future work.

User study. Although we have devised objective measures of quality for intent inference, the perceived quality is determined by the end-user. Therefore, another important direction for future work is to perform user studies on the usefulness of Anime for network operators.

6.1.3 Modeling and P4 Analysis

Control and data plane co-verification. In addition to P4, the semantics of major programming languages including C and Java have already been defined in the K framework. This provides us with an opportunity to combine the semantics of controller programs written in these languages with our P4 semantics of data plane and analyze a complete model of the whole network. This would be particularly interesting since the current P4 analysis tools solely focus on the data plane. If done, to the best of our knowledge, it would be the first tool to provide the ability to co-verify control and data plane based on a precise model of the two planes (not just simple abstraction of them).

Real-world compiler translation-validation In this dissertation, we showed the basics of translation validation with a simple P4 program that is manually translated into a toy language. We plan to provide tooling for translation validation of any P4 program using a real-world compiler of P4. We would need to define the semantics of the target language in K (in case it is not already defined) and also develop a method to automatically generate the synchronization points. Our initial guess is that the 4 synchronization points described in § 4.4.6 would be enough for the equivalence-checker to establish the equivalence between a P4 program and its compilation into any other language (if the compilation is correct).

Compiler conformance test suite generation Another interesting direction is to automatically generate a suite of test cases that cover all of the semantic rules in P4K. This would be particularly useful for the community since potentially many different P4 compilers will emerge in the future, each targeting a specific type of programmable device. It is therefore important to have a conformance test suite that makes sure the compilers adhere to the language specification. Currently, the K framework does not provide such a functionality, but we believe it is possible to implement the functionality for the framework.

6.1.4 Vision: Verified Infrastructure

Our work touches only one (although important) part of the bigger picture in the reliability of networked systems. In this dissertation, we have been mostly focusing on the data

plane. In addition, network control plane verification has received some attention in the literature (e.g., [10, 11, 12, 142]). However, a modern service infrastructure typically consists of multiple automated or semi-automated dynamic control components working at various layers (including layers beyond networking) providing a broad spectrum of service- and network-centric functions. For example, the scheduler in an orchestration system such as Kubernetes [143] controls the placement of application containers according to server resources, while a load-balancer (e.g., NGINX) manages the amount of traffic sent to each application instance according to end-to-end request latency. In the meantime, various routing and traffic engineering mechanisms (e.g., BGP, ECMP, MPLS-TE) manage network connectivity and performance. Non-trivial effects emerge from such a diverse range of control components interacting with each other and with the environment. This complexity paves the way for a range of failures that may only manifest under a certain combination of non-deterministic interactions, making them hard to detect before deployment. The existing infrastructure verification and network verification tools are a poor fit for this emerging space. They either solely focus on the networking layer, ignoring higher-level control components like orchestration systems [1, 5, 11, 86, 100, 118, 138, 139, 140, 144, 145]; only consider logical properties like network reachability rather than quantitative ones (e.g., load, latency) [11, 100, 144, 145]; focus on static snapshots rather than dynamic control [138, 140]; target specific protocols (e.g., BGP, ECMP) [146]; or focus on low-level system details rather than inter-component interactions (e.g. idempotency of provisioning scripts) [147]. In our recent work [148] we call for attention to the need for a formal understanding of dynamic infrastructure control. One interesting and challenging future direction would be to lay the formal foundations for verification of systems in this emerging space. This calls for a revision of our prior assumptions and approaches towards the three ingredients of verification.

References

[1] A. Horn, A. Kheradmand, and M. R. Prasad, "A precise and expressive lattice-theoretical framework for efficient network verification," in *27th IEEE International Conference on Network Protocols, ICNP 2019, Chicago, IL, USA, October 8-10, 2019.* IEEE, 2019, pp. 1–12.

[2] A. Kheradmand, "Automatic inference of high-level network intents by mining forwarding patterns," in *SOSR '20: Symposium on SDN Research, San Jose, CA, USA, March 3, 2020.* ACM, 2020, pp. 27–33.

[3] A. Kheradmand and G. Rosu, "P4K: A formal semantics of P4 and applications," *CoRR*, vol. abs/1804.01468, 2018. [Online]. Available: http://arxiv.org/abs/1804.01468

[4] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *SIGCOMM*. ACM, 2016, pp. 328–341.

[5] A. Horn, A. Kheradmand, and M. R. Prasad, "Delta-net: Real-time network verification using atoms," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017.* USENIX Association, 2017, pp. 735–749.

[6] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI*, 2013.

[7] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, vol. 12, 2012, pp. 113–126.

[8] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *ICNP*, 2013.

[9] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013, pp. 99–111.

[10] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017.* ACM, 2017, pp. 155–168.

[11] S. Prabhu, K. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020.* USENIX Association, 2020, pp. 953–967.

[12] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *NSDI*, 2020.

[13] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards verifying controller programs in software-defined networks," in *PLDI*, 2014.

[14] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese, "ddnf: An efficient data structure for header spaces," in *Haifa Verification Conference*. Springer, 2016, pp. 49–64.

[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*. ACM, 2011, pp. 290–301.

[16] Barefoot Networks, "Tofino 2, second-generation of world's fastest p4-programmable ethernet switch asics," https://www.barefootnetworks.com/products/brief-tofino-2/, 2020.

[17] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in p4 networks," Microsoft Research, Tech. Rep., 2016.

[18] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *SIGCOMM*, 2018.

[19] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *SIGCOMM*, 2018.

[20] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *SOSR*, 2018.

[21] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in p4 programs with assertion-based verification," in *SOSR*, 2018.

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[23] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010, http://kframework.org/.

[24] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge university press, 1999.

[25] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[26] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[27] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on computers*, vol. 27, no. 06, pp. 509–516, 1978.

[28] Web Application Firewall example, https://www.wafcharm.com/en/blog/how-to-block-a-request-that-contains-a-specific-string-in-uri-with-aws-waf/.

[29] A. Horn, A. Kheradmand, and M. R. Prasad, "A precise and expressive lattice-theoretical framework for efficient network verification," https://arxiv.org/abs/1908.09068, Tech. Rep., August 2019.

[30] D. G. Kourie, S. Obiedkov, B. W. Watson, and D. van der Merwe, "An incremental algorithm to construct a lattice of set intersections," *Science of Computer Programming*, vol. 74, no. 3, pp. 128–142, 2009.

[31] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle, "Verified iptables firewall analysis," in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2016, pp. 252–260.

[32] J. Stringer, D. Pemberton, Q. Fu, C. Lorier, R. Nelson, J. Bailey, C. N. Corrêa, and C. E. Rothenberg, "Cardigan: Sdn distributed routing fabric going live at an internet exchange," in *2014 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2014, pp. 1–7.

[33] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *NSDI*, 2015.

[34] Route Views, http://www.routeviews.org/.

[35] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008.

[36] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[37] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *SOSR*, 2016.

[38] A. Khurshid and B. Godfrey, personal communication, Jan. 2019.

[39] A. Khurshid, personal communication, Aug. 2019.

[40] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *POPL*, 2014, pp. 113–126.

[41] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[42] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 29–42.

[43] R. Birkner, D. Drachlser-Cohen, L. Vanbever, and M. Vechev, "Net2Text: Query-Guided Summarization of Network Forwarding Behaviors," in *USENIX NSDI*, Renton, WA, USA, 2018.

[44] A. Horn and A. Kheradmand, "Network analysis, US patent 10,439,926 B2," October 2019.

[45] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Config2spec: Mining network specifications from network configurations," in *USENIX NSDI*, 2020.

[46] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *NSDI*, 2015.

[47] S. Saha, S. Prabhu, and P. Madhusudan, "Netgen: Synthesizing data-plane configurations for network policies," in *SOSR*. ACM, 2015, p. 17.

[48] S. Prabhu, M. Dong, T. Meng, P. Godfrey, and M. Caesar, "Let me rephrase that: Transparent optimization in sdns," in *SOSR*, 2017, pp. 41–47.

[49] W. Zhou, J. Croft, B. Liu, and M. Caesar, "Neat: Network error auto-correct," in *SOSR*, 2017, pp. 157–163.

[50] A. Kheradmand, "Anime github repository," 2020. [Online]. Available: https://tinyurl.com/anime-supp-sosr20

[51] F. Murtagh and P. Legendre, "Ward's hierarchical agglomerative clustering method: which algorithms implement ward's criterion?" *Journal of classification*, vol. 31, no. 3, pp. 274–295, 2014.

[52] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[53] A. Papadopoulos and Y. Manolopoulos, "Performance of nearest neighbor queries in r-trees," in *International Conference on Database Theory*. Springer, 1997, pp. 394–408.

[54] A. Kheradmand, "Anime supplemental material," 2020. [Online]. Available: https://tinyurl.com/anime-supp-sosr20

[55] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," *Communications of the ACM*, vol. 58, no. 11, pp. 90–99, 2015.

[56] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[57] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, pp. 18–24, 2016.

[58] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *OOPSLA*. ACM, Nov 2016, pp. 74–91.

[59] P4 Language Consortium, "$P4_{14}$ language specification version 1.0.4," https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf, May 2017.

[60] A. Kheradmand, "P4 languagee specification repository issue #398," https://github.com/p4lang/p4-spec/issues/398, 2017.

[61] A. Kheradmand, "P4 languagee specification repository issue #411," https://github.com/p4lang/p4-spec/issues/411, 2017.

[62] A. Kheradmand, "P4 languagee specification repository issue #412," https://github.com/p4lang/p4-spec/issues/412, 2017.

[63] A. Kheradmand, "P4 languagee specification repository issue #414," https://github.com/p4lang/p4-spec/issues/414, 2017.

[64] A. Kheradmand, "P4 languagee specification repository issue #429," https://github.com/p4lang/p4-spec/issues/429, 2017.

[65] A. Kheradmand, "P4 languagee specification repository issue #430," https://github.com/p4lang/p4-spec/issues/430, 2017.

[66] A. Kheradmand, "P4 languagee specification repository issue #431," https://github.com/p4lang/p4-spec/issues/431, 2017.

[67] A. Kheradmand, "P4 languagee specification repository issue #433," https://github.com/p4lang/p4-spec/issues/433, 2017.

[68] A. Kheradmand, "P4 languagee specification repository issue #442," https://github.com/p4lang/p4-spec/issues/442, 2017.

[69] A. Kheradmand, "Suggested revision for $P4_{14}$ language specification," https://github.com/kheradmand/p4-spec, 2017.

[70] P4 Language Consortium, "P4 reference compiler (p4c)," https://github.com/p4lang/p4c, 2017.

[71] C. Ellison and G. Roşu, "An executable formal semantics of c with applications," in *POPL*. ACM, January 2012, pp. 533–544.

[72] D. Bogdănaş and G. Roşu, "K-Java: A Complete Semantics of Java," in *POPL*. ACM, January 2015, pp. 445–456.

[73] D. Park, A. Ştefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *PLDI*. ACM, June 2015, pp. 346–356.

[74] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[75] P4 Language Consortium, "$P4_{16}$ language specification version 1.0.0," https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf, May 2017.

[76] A. Fingerhut, "Operations on header stacks in $P4_{14}$, $P4_{16}$, and bmv2," https://github.com/jafingerhut/p4-guide/blob/master/README-header-stacks.md, 2017.

[77] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, D. Guth, P. Daian, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," University of Illinois at Urbana-Champaign, Tech. Rep., 2017.

[78] B. S. Davie and Y. Rekhter, *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc., 2000.

[79] P4 Language Consortium, "$P4_{14}$ language specification version 1.0.3," https://p4lang.github.io/p4-spec/p4-14/v1.1.0/tex/p4.pdf, January 2016.

[80] P4 Language Consortium, "$P4_{14}$ language specification version 1.0.3," https://p4lang.github.io/p4-spec/p4-14/v1.0.3/tex/p4.pdf, November 2016.

[81] D. Guth, C. Hathhorn, M. Saxena, and G. Roşu, "Rv-match: Practical semantics-based program analysis," in *CAV*, ser. LNCS, vol. 9779. Springer, July 2016, pp. 447–453.

[82] A. Bas, "Basic routing example," https://github.com/p4lang/p4factory/tree/master/targets/basic_routing, 2016.

[83] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *TACAS*, 1998, pp. 151–166.

[84] K Framework Development Team, "Keq," https://github.com/kframework/k/tree/keq, 2017.

[85] P4 Language Consortium, "switch.p4," https://github.com/p4lang/switch, 2016.

[86] S. Prabhu, A. Kheradmand, B. Godfrey, and M. Caesar, "Predicting network futures with plankton," in *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*. ACM, 2017, pp. 92–98.

[87] T. G. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *SIGCOMM*, 1999.

[88] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *NSDI*, 2005, pp. 43–56.

[89] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with C-BGP," *IEEE Network*, vol. 19, no. 6, Nov. 2005.

[90] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott, "FSR: Formal analysis and implementation toolkit for safe interdomain routing," *IEEE/ACM Transactions on Networking*, vol. 20, no. 6, Dec. 2012.

[91] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Formal semantics and automated verification for the border gateway protocol," in *NetPL*, 2016.

[92] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. D. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *OSDI)*, 2016.

[93] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *SIGCOMM*, 2016, pp. 300–313.

[94] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," *ACM Transactions on Information and System Security*, vol. 14, no. 1, June 2011.

[95] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin, "Automatic error finding in access-control policies," in *CCS*, 2011.

[96] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for firewall modeling and analysis," in *SP*, 2006.

[97] A. Jeffrey and T. Samak, "Model checking firewall policy configurations," in *POLICY*, 2009.

[98] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave tool for firewall analysis," in *LISA*, 2010.

[99] S. Zhang, A. Mahmoud, S. Malik, and S. Narain, "Verification and synthesis of firewalls using sat and qbf," in *ICNP*, 2012.

[100] M. Canini, D. Venzano, P. Peresíni, D. Kostic, and J. Rexford, "A NICE way to test openflow applications," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012.* USENIX Association, 2012, pp. 127–140.

[101] S. Son, S. Shin, V. Yegneswaran, P. A. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *ICC*, 2013.

[102] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese, "Correct by construction networks using stepwise refinement." in *NSDI*, 2017.

[103] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *CoNEXT*, 2012.

[104] D. Lebrun, S. Vissicchio, and O. Bonaventure, "Towards test-driven software defined networking," in *NOMS*, 2014.

[105] M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, "Chaos monkey: Increasing sdn reliability through systematic network destruction," in *SIGCOMM*, 2015.

[106] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *NSDI*, 2016.

[107] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully emulating large production networks," in *SOSP*, 2017.

[108] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, "An assertion language for debugging SDN applications," in *HotSDN*, 2014.

[109] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *SIGCOMM*, 2014.

[110] T. Nelson, A. D. Ferguson, and S. Krishnamurthi, "Static differential program analysis for software-defined networks," in *FM*, 2015.

[111] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, "SDNRacer: Detecting concurrency violations in software-defined networks," in *SOSR*, 2015.

[112] R. May, A. El-Hassany, L. Vanbever, and M. Vechev, "BigBug: Practical concurrency analysis for SDN," in *SOSR*, 2017.

[113] H. Hojjat, P. Rümmer, J. McClurg, P. Černỳ, and N. Foster, "Optimizing Horn solvers for network repair," in *FMCAD*, 2016.

[114] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu, "Automatically repairing network control planes using an abstract representation," in *SOSP*, 2017.

[115] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks." in *NSDI*, 2017.

[116] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *CAV*, 2017.

[117] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent NetCore: From policies to pipelines," in *ICFP*, 2014.

[118] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark, "Kinetic: Verifiable dynamic network control," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 2015, pp. 59–72.

[119] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[120] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford, "HotSwap: Correct and efficient controller upgrades for software-defined networks," in *HotSDN*, 2013.

[121] S. Vissicchio, L. Vanbever, L. Cittadini, G. G. Xie, and O. Bonaventure, "Safe update of hybrid SDN networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, June 2017.

[122] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in SDN," in *SOSR*, 2017.

[123] G. G. Xie, J. Zhanm, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of IP networks," in *INFOCOM*, 2005.

[124] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *NSDI*, 2015.

[125] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *NSDI*, 2014.

[126] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. El-Badawi, "Network configuration in a box: towards end-to-end verification of network reachability and security," in *ICNP*, 2009.

[127] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *SafeConfig*, 2010.

[128] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," Microsoft Research, Tech. Rep., 2014.

[129] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *POPL*, 2016.

[130] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *SIGCOMM*, 2018.

[131] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[132] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of c," in *PLDI*. ACM, June 2015, pp. 336–345.

[133] Ecma TC39, "Standard ECMA-262 ECMAScript Language Specification Edition 5.1," June 2011.

[134] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014.

[135] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *SIGCOMM*, 2016.

[136] C. Cadar, D. Dunbar, D. R. Engler et al., "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008.

[137] A quotation from a 1942 speech by Winston Churchill.

[138] Y. Zhang, W. Wu, S. Banerjee, J. Kang, and M. A. Sánchez, "Sla-verifier: Stateful and quantitative verification for service chaining," in *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*. IEEE, 2017, pp. 1–9.

[139] A. Abhashkumar, J. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting diverse dynamic intent-based policies using janus," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*. ACM, 2017, pp. 296–309.

[140] G. Juniwal, N. Bjorner, R. Mahajan, S. Seshia, and G. Varghese, "Quantitative network analysis," *Technical report*, 2016.

[141] S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva, "Cantor meets scott: semantic foundations for probabilistic networks," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 2017, pp. 557–571.

[142] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 201–219.

[143] Kubernetes, "Kubernetes: Production-grade container orchestration," https://kubernetes.io/, June 2020.

[144] Y. Yuan, S. Moon, S. Uppal, L. Jia, and V. Sekar, "Netsmc: A custom symbolic model checker for stateful network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 181–200.

[145] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 257–272.

[146] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 2020, pp. 974–988.

[147] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: a configuration verification tool for puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 2016, pp. 416–430.

[148] B. Liu, A. Kheradmand, M. Caesar, and B. Godfrey, "Towards verified self-driving infrastructure," in *HotNets*, 2020.